HI-RES GRAPHICS FOR THE APPLE® II / IIe / IIc

ROBERT L. TICE &
DOROTHY JO STEVENS

No. 1849
$21.95

HI-RES GRAPHICS FOR THE APPLE® II / IIe / IIc

ROBERT L. TICE & DOROTHY JO STEVENS

TAB TAB BOOKS Inc.
Blue Ridge Summit, PA 17214

Apple, Apple II+, Applesoft, and Apple //e are registered
trademarks of Apple Computer Inc.

FIRST EDITION

FIRST PRINTING

Copyright © 1985 by TAB BOOKS Inc.
Printed in the United States of America

Reproduction or publication of the content in any manner, without express
permission of the publisher, is prohibited. No liability is assumed with respect to
the use of the information herein.

Library of Congress Cataloging in Publication Data

Tice, Robert L.
Hi-res graphics for the Apple II//e//c.

Includes index.
1. Computer graphics. 2. Apple II (Computer)—
Programming. 3. Apple //e (Computer)—Programming.
4. Apple //c.(Computer)—Programming. I. Stevens,
Dorothy Jo. II. Title.
T385.T52 1985 001.64'43 84-23954
ISBN 0-8306-0849-4
ISBN 0-8306-1849-X (pbk).

Cover concept by Ken Jensen, Design Center, University of Nebraska-Lincoln.

Contents

Preface v

Introduction vi

1 Overview 1
Program Presentations—Notation Conventions—Required Equipment

2 Low-Resolution Graphics 3
Low-Resolution Graphics Screen—The Low-Resolution Instruction Set—Designing Low-Resolution Pictures—Implementing the Instruction Set—Random Graphics Displays—Integrating Sound with Low-Res Graphics—Applications

3 High-Resolution Graphics 26
High-Resolution Instruction Set—Properties of High-Resolution Graphics—Screen Management—High-Resolution Graphics Applications—Page Flipping

4 Vector Graphics 59
Designing Shapes—Shading a Selected Design or Shape—Tracing the Vector Path—Coding the Vectors—Charting the Codes—Coding the Chart Contents—Constructing a Shape Table—Entering and Saving a Shape Table

5 Using Shape Tables in BASIC Programs 74
Displaying Shapes on the Screen—Shape Table Related Commands—Demonstration Programs—Mixing Text and Graphics—Arcade-Style Programs—Program Ideas—Helpful Hints

6 Shape Table Utility Programs 108
Table Developer—Table Merger—Shapely Scenes

7 Music and Sound 124
Simple Sounds—Music and Songs—Arcade Game Sounds

8 Combining High-Resolution Graphics Techniques 143
Vector Lines and Designs—Plot Graphics Enhancements of Vector Graphics—Utilizing Polar Equations, Shapes, and Sounds

9 Machine Language Topics 165
Displaying Shapes in Machine Language—Raster Graphics

Appendix A ASCII Codes 174

Appendix B Computer Number Systems 176

Appendix C Sample Shape Table Codes 178

Appendix D Computer Memory Structures 184

Appendix E Computer Memory Utilization 185

Appendix F Summary of Programs 189

Index 193

Preface

This book uses a step-by-step approach to explain the fundamentals and advanced techniques of graphics. The sequencing of topics and the programs selected to demonstrate those concepts are based on experiences in commercial software design and from teaching many students ranging from junior and senior high school students to adults. Because students were extremely receptive to materials and examples used in Dr. Stevens' computing courses and respected the commercial success of Mr. Tice, we made the decision to combine these two levels of expertise in one book–this book.

BASIC has become a common language that has evolved, over a number of years, from a language intended mainly for student use to a relatively sophisticated language often used for large-scale development projects. In addition, many educational institutions and agencies, large numbers of commercial software development concerns, and personal computer owners use Apple computers. Thus we decided to focus on BASIC and the programming commands used with Apple computers. It seemed logical to concentrate on one brand of computer—and do it well—rather than weaken and interrupt the flow of content by attempting to include the graphics commands of several computers. Once the concepts are learned, however, then it is very easy to transfer those concepts to other brands of computers.

Topics are presented to teach graphics as a functional tool for software development—whether the user is a junior or senior high student, teacher, hobbyist, or commercial programmer desiring to jazz up a program with attractive displays, motion, and sound. Concepts presented are appropriate for courses, workshops, seminars, enrichment topics for programmers, supplementary material to computer courses, and computer camps, as well as for independent study. Whether there is enough content for a one- or two-semester course depends on the programming expertise of the participants and the objectives of the users.

Introduction

Although this book uses a step-by-step format, students are able to learn rapidly because the examples are specifically designed to bridge directly with the concepts being presented. Furthermore, the examples used can be related to various interests and often can be integrated, as is, into a variety of programs. This is true whether the concepts presented are simple BASIC commands or complex high-resolution graphics pictures using shape tables with motion and sound effects. We made a deliberate effort to blend examples, program descriptions, and utility programs to allow you to assimilate concepts commensurate with your level of expertise and your reasons for wanting to learn graphics. If you are just beginning to learn about graphics, you will find the detailed explanations of initial BASIC commands very helpful; if you are an experienced programmer, you may also benefit from this review of BASIC commands. Not only do you learn how to use advanced, efficient graphics techniques, but your initial knowledge of BASIC commands is also enhanced.

CONTENT ORGANIZATION

The first chapter is essentially an introduction with recommendations on how to use the book. The book is organized so that you may begin working with the computer as soon as possible. Chapter 2 contains material that involves you immediately. Low-resolution graphics are presented, sounds (such as buzzes and clicks) that can be controlled from BASIC are introduced, and all of the concepts are then incorporated into the Pink Out game at the conclusion of the chapter. Information on high-resolution graphics is presented in Chapter 3. Initial instruction begins with plotting points and moves quickly to drawing lines, figures, solid polygons, complex polar designs, and graphing equations. The highlight of this chapter is a utility mathematics program that provides all of the mathematics necessary for using some very mathematically complex programs. Consequently, little or no knowledge of mathematics is required. The chapter concludes with all of the concepts integrated into a Winter Word Game program.

Chapter 4 concentrates on the design and placement of shapes and how to save shapes on a disk. You will gain a depth of understanding in coding, translating, and saving shapes on disks.

Shape tables are presented in Chapter 5. Topics on creating motion, generating character sets, and mixing text and high-resolution graphics are explained and demonstrated. Numerous demonstrations reinforce the concepts of scaling, rotation, and color. The chapter concludes with a snappy arcade game.

Music and sounds are presented in Chapter 7. A sound effects utility program is very useful to programmers is presented. For example, Song Editor allows complete songs to be played; all you have to do is to type in the notes. The unique part is that Song Editor allows notes to be held longer than conventional music programs. Gaming sounds such as a laser sound are explained and demonstrated. Your imagination can really take over at this point. Concepts presented in this chapter allow for a multitude of sounds to be generated, ranging from the sound of a starting car to the sounds of an intergalactic war.

Chapter 8 combines concepts. Our intent here was not to present ideas as disjoint sets, but to encourage interesting combinations of ideas with practical or entertaining results. An excellent example is provided with the Fraction Action educational game. This program integrates high-resolution graphics, vectors, motion, music, and sounds into a clever and worthwhile, yet entertaining, educational learning experience. When this program was tested with groups of junior high students the results were amazing. In this chapter page flipping is used to create motion. You are urged to transfer these concepts to your other areas of interests. For example, the circles in this Fraction Acton program would be ideal for pie charts in a business program.

Chapter 9 is a short but interesting chapter with a different sort of ending. Programs are selected to compare the effects of using shapes from BASIC using page flipping and without page flipping to those written in machine language using page flipping and without page flipping. The primary objective is to make you aware of the advantages of learning about machine language. Many persons believed machine language to be difficult; this brief exposure may motivate you to explore and learn about raster graphics or bit mapping techniques.

While each chapter is independent of the others, you are encouraged to read the chapters sequentially. Beneficial tips and ideas are presented throughout the book. For example, paddles are presented only in Chapter 2; if that chapter is skipped then the paddle concept is missed.

Each chapter contains a description of common programming errors that may occur, as well as hints for debugging. Wherever appropriate, concepts that require conversion tables or similar types of references, appendices have been provided.

1

Overview

The Apple II computer has excellent color graphics. There are three methods that can be used to create graphic displays: plot, vector, and raster (bit mapping). This book explains both the plot and vector graphics methods.

Ideas are presented in this chapter to help guide you in your quest to learn grapics programming techniques. We will discuss the following topics in Chapter 1.

1. Using this book.
2. Program presentations
3. Notation conventions.
4. Required equipment.

The format of this book is such that you can learn graphics gradually and at your own pace. Material presented becomes increasingly advanced as each chapter progresses. *Hi-Res Graphics for the Apple* II /II e /II c is designed to be useful to individuals with various levels of programming expertise. Naturally, some knowledge of BASIC is advantageous. The information is presented in such a way, however, that beginning programmers can learn to utilize most of the graphics routines demonstrated. More advanced programmers will be pleasantly surprised to find programming techniques illustrated that are coveted by most commercial programmers but are rarely shared with others.

You must take an active role in the learning process in order to master the techniques presented in this book. Computer programming cannot be learned by simply reading a book. The programs presented should be entered into the computer and executed. The monitor screen should be observed closely while the designs are displayed. If you find yourself asking, "Why did it do that?" referring to the text to seek the answers will help you gain the most from reading this book.

A conscientious effort was made to avoid the use of needless computer jargon whenever possible. However, many ideas are best conveyed with terminology uniquely associated with computers and computer programming. Brief explanations are given for words used in this book that have special meanings. Whenever detailed discussions are required to clarify these concepts, we refer you to the appropriate Appendix so that the flow of the chapter is not disrupted.

Every chapter should be read, regardless of your level of programming expertise. Skipping a chapter may cause you to miss explanations of programming techniques that are useful for all graphics modes. While critical information is repeated several times to assist you, some specific programming techniques are presented in only one chapter.

PROGRAM PRESENTATIONS

A diverse selection of programs is used to demonstrate the concepts presented in this book. The programs are designed to accommodate different interests of readers. Illustrated here are graphics programming techniques that can be used in the design of games, educational software, computer art, and many other types of programs.

Several approaches are employed to explain program statements. Line-by-line discussions of most of the programs are presented to help you understand every detail of the program. Pitfalls and errors commonly made during initial attempts to use graphics are discussed. To further meet the needs of beginning programmers, most BASIC statements are explained the first time they appear in a program. This form of explanation allows you to become comfortable with each programming skill presented before moving to the next concept.

Another technique used to help you understand graphics programming is to present the reasoning used in determining the functions that a module or segment of a program should perform. This presentation format allows you to examine the processes involved in planning and developing efficient programs.

Some of the programs listed in this book are designed to be used as utility programs. These programs are intended to reduce the amount of time and effort you most devote to developing interesting graphics displays. While instructions for operating and using the programs are provided, line-by-line discussions of the programs are not presented. The programs can be used effectively without investigating the programming techniques used, and those of you who decide to study the program listings will generally recognize that the techniques used in these programs are discussed in other sections of the book.

NOTATION CONVENTIONS

Certain conventions with respect to program discussions have been adopted and are used throughout the book. The names of the variables used in demonstration programs are generally based on one or two letters from the description of the variable's role. When this role is described, the letter or letters used in formulating the variable name are capitalized. For example, if the variable NP is to be used to store the number of players playing a game, then the description of the variable's role would appear as *Number of Players*.

In order to help you distinguish between variables used in a specific programming statement and those used to illustrate general forms of a BASIC command or mathematical formula, we decided to use uppercase letters for program variable names and to use lowercase letters to illustrate general-form variables.

REQUIRED EQUIPMENT

All the programs presented in this book were developed using Applesoft BASIC on a standard Apple II+ computer with 48K of RAM. Most of the applications discussed can be accomplished on a computer with a single disk drive. Either the Apple II+ or Apple *II*e can be used for testing and studying the programs in *Hi-Res Graphics for the Apple*. A color monitor and paddles or joysticks are recommended, but are not required for most of the topics covered.

Although many printers have graphics printing capability, printer applications are not discussed in this book. Readers who want to use a printer to output graphics will find that adequate information is included in the documentation that accompanies most printers and interface cards having graphics output capabilities.

2

Low-Resolution Graphics

The graphics displays called *plot graphics* are created by instructing the computer to light or color particular points on the display screen. Apple computers allow users to create plot graphics displays with two sizes of points. The larger the points, the fewer that are available on the screen. The fewer the points available on the screen, the lower the *resolution*. The Apple's plot graphics are divided into two categories, *high-resolution graphics* and *low-resolution graphics*. Both graphics resolutions cannot be displayed at the same time on the screen, and a separate instruction set is needed for each. This chapter covers low-resolution graphics and topics that are relevant to low-resolution graphics programming:

1. Low-resolution graphics screen.
2. Low-resolution graphics instruction set.
3. Designing low-resolution graphics pictures.
4. Implementing the low-resolution graphics instruction set.
5. Random low-resolution graphics displays.
6. Integrating sound with low-resolution graphics.
7. Application of low-resolution graphics.

The Apple II computer has three screen display modes. Words and numbers are displayed in the TEXT mode. That's good, because words and numbers are the primary components of text. Two other modes are used for graphics displays, LOW-RESOLUTION mode and HIGH-RESOLUTION mode. Each display mode uses the screen in different ways. In order to understand how low-resolution graphics displays are created, it is necessary to understand the utilization of the screen in this display mode.

LOW-RESOLUTION GRAPHICS SCREEN

The low-resolution graphics display mode allows you to create colorful displays. There are 16 colors available using this type of plot graphics (see Table 2-1). Some sacrifices must be made in order to offer such an array of colors. Certain *pixels* (pictures elements) on the color monitor are capable of

Table 2-1. Low-Resolution Colors.

Code	Color	Code	Color
0	black	8	brown
1	magenta	9	orange
2	dark blue	10	grey
3	purple	11	pink
4	dark green	12	green
5	grey	13	yellow
6	medium blue	14	aqua
7	light blue	15	white

generating particular colors. In order to create many different colors, it is necessary to utilize adjacent pixels. The greater the choice of colors, the more pixels that are needed to create or generate the color, and the large number of pixels necessary to provide the low-resolution colors reduces the number of points available on the screen to 1600. Each low-resolution point is 7 pixels wide and 4 pixels high.

The low-resolution screen can display 40 points horizontally (columns) and 40 points vertically (rows). Each column of the screen is assigned a unique number from 0 through 39. The left-most column on the screen is column 0 and the right-most column is column 39. The rows are numbered from the top of the screen (row 0) to the bottom (row 39). Any point can be referenced by its column and row positions on the screen.

The number which corresponds to the column of the screen in which the point is located is called the *horizontal coordinate* of the point. The *vertical coordinate* of a point is the number corresponding to the screen row in which the point is located. Any point on the screen thus can be referenced by a pair of numbers. Since the numbers will always be stated in the order *horizontal coordinate, vertical coordinate,* it is called an *ordered pair.* The ordered pair which describes the location of any point on the screen are said to be the *coordinates* of that point (see Fig. 2-1).

The computer does not normally use the entire video screen for low-resolution graphics displays. Four lines at the bottom are reserved for displaying textual information. This allows programmers enough room to PRINT instructions to the user, INPUT user responses, or to describe the graphics display.

THE LOW-RESOLUTION INSTRUCTION SET

The Applesoft BASIC instructions or commands used for creating low-resolution displays are introduced and explained below.

GR. GR is the command that instructs the computer that it is to use the low-resolution graphics mode for displaying information on the screen. When the command is used, the computer erases anything that is displayed on the low-resolution graphics portion of the screen (the top 20 text lines). This command must be executed before low-resolution graphics points can be plotted.

COLOR. The color that is to be used for plotting a point on the screen must be specified prior to plotting the point the syntax for the COLOR command is

COLOR = {n}

where n is a number from 0 through 15 corresponding to the color desired. For example, COLOR=3 causes plotted points to be purple.

PLOT. Any point on the low-resolution screen can be plotted using the PLOT command. The command must be followed by the coordinates of the point to be plotted. The general form of the command is

PLOT {x,y}

where x and y are both numbers from 0 to 39. X represents the horizontal position of the point and y represents its vertical position. The statement PLOT 20,20 causes the center point of the low-resolution screen to be plotted using the most recently stated color.

HLIN. Whenever consecutive horizontal points are to be plotted, the HLIN command can be used. The syntax for using the Horizontal LINe command is

HLIN{x1,x2} AT{y}

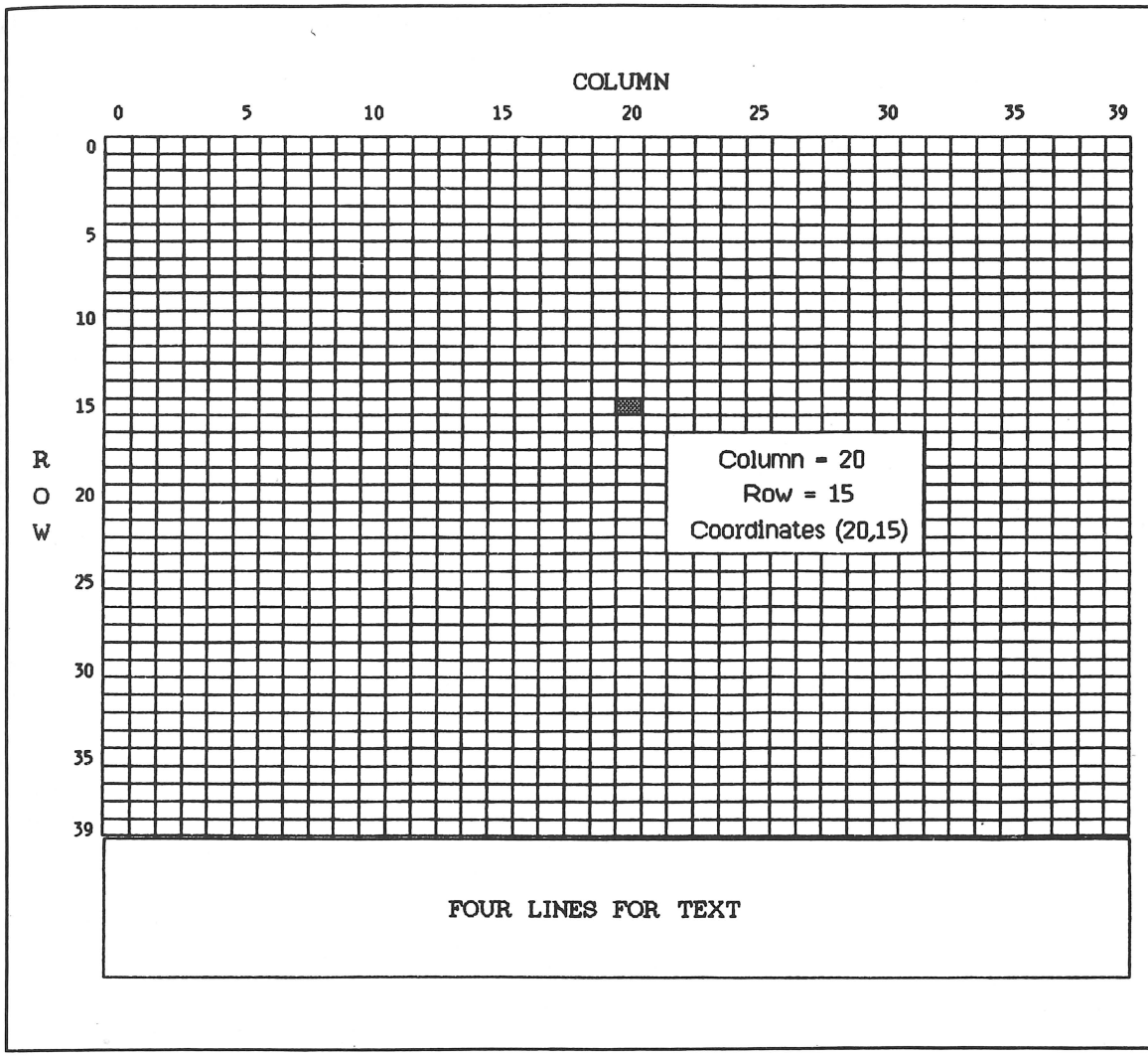


Fig. 2-1. The positions of points on the low-resolution graphics screen are identified by their column and row coordinates.

the term x1 specifies one horizontal endpoint of the line, x2 specifies the other endpoint, and y specifies the vertical coordinate of each point on the line. Legal values for x1, x2, and y are 0 to 39. In order to plot a horizontal line across the top of the screen, use the command HLIN 0,39 AT 0. A horizontal line across the left half of the screen on the bottom row is plotted using the command HLIN 0,20 AT 39.

VLIN. This command is similar to HLIN, except it is used to plot Vertical LINes. The general form for the VLIN command is

VLIN{y1,y2} AT{x}

where y1 and y2 are the endpoints of the vertical line and x is the horizontal coordinate for each point on the line. A vertical line extending from the middle of the screen to the top of the screen at the 10th horizontal coordinate is plotted when the following statement is executed: VLIN 20,0 AT 10.

TEXT. While TEXT is not a graphics command, it is included because it is needed to return to the text display mode after graphics have been used.

When TEXT is executed to terminate use of the low-resolution graphics screen, the computer places inverse video @ characters and other equally distracting images on the top 20 rows of the screen. The TEXT command should be followed immediately by the HOME command, which clears the screen.

There is not a great number of low-resolution commands, but interesting displays can be achieved when the above commands are used in conjunction with other BASIC statements.

DESIGNING LOW-RESOLUTION PICTURES

As is the case with many Applesoft BASIC commands, the low-resolution graphics commands can be used in *immediate mode.* This simply means that the computer can execute the commands immediately if they are entered without line numbers. The statements used in the previous section are presented here to demonstrate the use of graphics commands in immediate mode. After the computer is operating, enter the following statements one at a time, just as they appear below, and observe the display before entering the subsequent statement.

```
GR
COLOR = 3
PLOT 20,20
HLIN 0,39 AT 0
COLOR = 5
HLIN 0,20 AT 39
COLOR = 1
VLIN 20,0 AT 10
TEXT
HOME
```

These commands are generally used from within an Applesoft program. When line numbers are assigned to statements, they are not executed until the computer receives the RUN command. Since the computer defers execution of the statements until the RUN command is received, this is called *deferred mode.* The statements listed above can be executed in deferred mode by simply typing a line number prior to entering each command. The list of a program using the preceding commands appear in Listing 2-1.

Listing 2-1. Low-Res Commands.

```
5  REM   LOW RES COMMANDS
10  GR
20  COLOR= 3
30  PLOT 20,20
40  HLIN 0,39 AT 0
50  COLOR= 5
60  HLIN 0,20 AT 39
70  COLOR= 1
80  VLIN 20,0 AT 10
90  TEXT
100  HOME
```

The above program should be entered and run. Notice that the graphics are displayed in a flash, then removed from the screen. In order to leave the graphics displayed for a set amount of time, a time delay loop could be placed in the program. The Apple II can count to approximately 800 in a single second. Placing the following line in the program causes the graphics to be displayed for approximately three seconds.

```
85  FOR TIME = 1 TO 2400: NEXT TIME
```

Perhaps a better way of regulating the amount of time the graphics are left on the screen is by using the GET statement. GET causes the execution of the program to halt until a key is pressed. The command must be followed by a variable to which the character corresponding to the pressed key is assigned. A string variable should always be used with the GET command. If a real or integer variable is used instead, and the user presses a key other than a number key, a syntax error occurs. Insert the following line in the above program to allow the user to regulate the amount of time the graphic is displayed on the screen.

```
85  GET GO$
```

Careful planning plays a significant role in creating successful graphics displays. A very helpful planning aid is a piece of graph paper. Shading or coloring grids on paper is useful in determining which of the low-resolution graphics commands should be used to generate the picture. Figure 2-2 shows a picture of a boat shaded on graph paper. In order to simplify the planning process, the boat will be a solid color.

Once the picture has been drawn, the coordinates of the points must be determined. It is not necessary to find the coordinates of every point. Try to look for lines, either vertical or horizontal, in the figure. If a vertical or horizontal sequence of

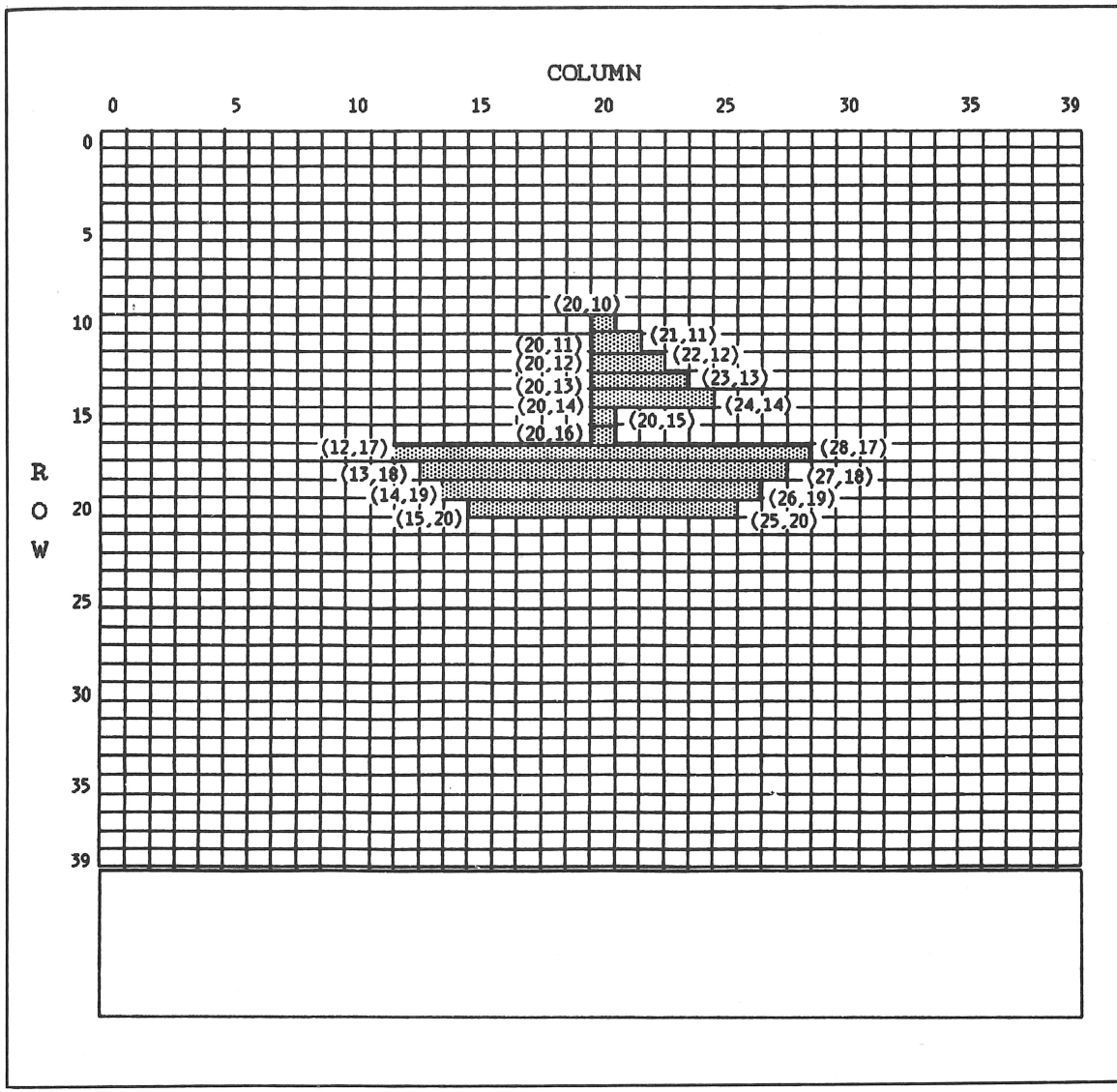


Fig. 2-2. The figure gives the coordinates of the endpoints of the horizontal lines and single points that were used to display the boat in the Purple Boat program.

Listing 2-2. Purple Boat.

```
5  REM     PURPLE BOAT
10  GR
20  COLOR= 3
500  HLIN 15,25 AT 20
510  HLIN 14,26 AT 19
520  HLIN 13,27 AT 18
530  HLIN 12,28 AT 17
540  PLOT 20,16
550  PLOT 20,15
560  HLIN 20,24 AT 14
570  HLIN 20,23 AT 13
580  HLIN 20,22 AT 12
590  HLIN 20,21 AT 11
600  PLOT 20,10
```

grids all require the same color, then only the coordinates of the endpoints need to be determined. Remember that lines can be plotted using the HLIN and VLIN commands. The coordinates of the points used to construct the boat are also presented in Fig. 2-2. Listing 2-2 is a program that displays the purple boat.

IMPLEMENTING THE INSTRUCTION SET

The above approach for programming a low-resolution graphics scene is straightforward and useful. However, it is often necessary to display a picture of an object in several areas of the screen. If a unique program segment were needed each time the figure was to appear in a different location on the screen, the work involved would discourage most programmers. In addition, programs using graphics would be very long. A more efficient programming technique is to use variables to describe the relative positions of the points needed to create the design.

This approach requires selecting one coordinate pair to be used to mark the figure's position on the screen. The point the coordinates describe is called the *reference point.* The reference point should be chosen carefully in order to minimize programming effort. Any point within the picture can serve as the reference point. The vertical coordinate of the lowest point on the boat (20) is used for the vertical coordinate in this example. While the horizontal coordinate most frequently used (also 20) is the horizontal coordinate of the reference point. Each of the selected coordinates should have a variable assigned to it. The variable X is used to represent the horizontal reference coordinate, and Y is used to represent the vertical reference coordinate. The ordered pair of coordinates for the reference point, (20,20), becomes (X,Y).

The next task is to describe the position of the remaining points in relation to the reference point. By inspecting the preceding program and Fig. 2-2, the coordinates of the points can be written in terms of X and Y. All the horizontal coordinates must be expressed in terms of X, and all the vertical coordinates must be expressed in terms of Y. For example, the statement HLIN 15,25 AT 20 must be changed to HLIN X−5,X+5 AT Y. Since X=20, X−5 is equivalent to 15 and X+5 equals 25. Y has a value of 20, so no addend is necessary.

A program that displays the boat using the reference point concept is listed as Listing 2-3.

Listing 2-3. Reference Point Boat.

```
5  REM   REFERENCE POINT BOAT
10  GR
20 X = 20
30 Y = 20
40  COLOR= 3
500  HLIN X - 5,X + 5 AT Y
510  HLIN X - 6,X + 6 AT Y - 1
520  HLIN X - 7,X + 7 AT Y - 2
530  HLIN X - 8,X + 8 AT Y - 3
540  PLOT X,Y - 4
550  PLOT X,Y - 5
560  HLIN X,X + 4 AT Y - 6
570  HLIN X,X + 3 AT Y - 7
580  HLIN X,X + 2 AT Y - 8
590  HLIN X,X + 1 AT Y - 9
600  PLOT X,Y - 10
```

Using this programming technique simplifies the programming task, particularly when more than one boat is to be displayed on the screen at a time. This is also true if one boat is to be positioned at different

areas of the screen at different times.

The segment of the program that actually displays the boat becomes a subroutine with the addition of the RETURN command at the end of the graphics segment. Prior to using the subroutine, the values for the coordinates of the key point must be specified. Listing 2-4 is a program that displays the boat in three different areas of the screen at the same time. The END statement has been inserted at line 120 so that the subroutine is not executed without a GOSUB statement. If the END statement is omitted, a RETURN WITHOUT GOSUB error will be forthcoming.

Listing 2-4. Three Boats.

```
5  REM    THREE BOATS
10  GR
20 X = 20
30 Y = 20
40  COLOR= 3
50  GOSUB 500
60 X = 10
70 Y = 10
80  GOSUB 500
90 X = 8
100 Y = 39
110  GOSUB 500
120  END
500  HLIN X - 5,X + 5 AT Y
510  HLIN X - 6,X + 6 AT Y - 1
520  HLIN X - 7,X + 7 AT Y - 2
530  HLIN X - 8,X + 8 AT Y - 3
540  PLOT X,Y - 4
550  PLOT X,Y - 5
560  HLIN X,X + 4 AT Y - 6
570  HLIN X,X + 3 AT Y - 7
580  HLIN X,X + 2 AT Y - 8
590  HLIN X,X + 1 AT Y - 9
600  PLOT X,Y - 10
610  RETURN
```

A more complex color scheme can be used to display the boat by inserting COLOR statements in the subroutine. The program in Listing 2-5 displays the boat with a medium blue hull, an orange mast, and a white sail.

Listing 2-5. Rainbow Boat.

```
5  REM     RAINBOW BOAT
10  GR
20 X = 20
30 Y = 20
40  COLOR= 6
500  HLIN X - 5,X + 5 AT Y
510  HLIN X - 6,X + 6 AT Y - 1
520  HLIN X - 7,X + 7 AT Y - 2
530  HLIN X - 8,X + 8 AT Y - 3
535  COLOR= 9
540  PLOT X,Y - 4
550  PLOT X,Y - 5
555  COLOR= 15
560  HLIN X,X + 4 AT Y - 6
570  HLIN X,X + 3 AT Y - 7
580  HLIN X,X + 2 AT Y - 8
590  HLIN X,X + 1 AT Y - 9
600  PLOT X,Y - 10
```

RANDOM GRAPHICS DISPLAYS

Many graphics applications rely upon displaying figures at random positions on the screen. Interesting visual effects can be achieved by selecting screen coordinates randomly. The RND function allows programmers to select or generate random numbers. The general form of the RND command is

RND(1)

The number 1 is called the *argument* of the *function* RND. Arguments other than the number 1 may be used, but all positive arguments have the same result. Zero used for the argument results in duplicating the previously selected random number. This book assumes that 1 will be used for the argument of the RND function. Random numbers generated by the Apple II are nine-digit decimals between 0 and 1. The statement LET X = RND(1) assigns a value like .567139084 to the real variable X. Mathematical transformations must be applied to the value before it is useful for most applications.

A very nice formula for transforming random numbers into the desired range is

INT(N * RND(1) + F)

INT is a function that changes any real number to an integer. It has the effect of discarding the decimal portion of a positive real number. For example, the value of INT(3.67896) is 3. It should be remembered that the INT function always rounds *down*. The value of the variable N in the formula determines the size of the range from which the random numbers are selected. If N=5, then the number of unique random integers that can result is 5. The value of the variable F specifies the least integer in the range. When F=1, the first possible value of the random formula is 1.

When low-resolution coordinates are to be selected randomly, the appropriate formula values are N=40 and F=0. Since F=0, it may be omitted from the formula. In addition, since all coordinates are integer numbers (no decimal parts), the INT portion of the formula is not necessary. Whenever numbers are to be generated randomly for any application that does not recognize decimals, the formula for selecting appropriate values is

N * RND(1) + F

Low-resolution colors may also be selected randomly. Since 16 colors, starting with 0, are available, the values N=16 and F=0 will cause the RND formula to generate legal color values. The program in Listing 2-6 demonstrates the use of random numbers in ploting points randomly on the screen with random colors. The program contains an infinite loop, which causes it to run forever. The program can be stopped by holding down the CTRL key while pressing C.

Listing 2-6. Random Plotter.

```
5  REM     RANDOM PLOTTER
10  GR
20  COLOR= 16 *  RND (1)
30  PLOT 40 *  RND (1),40 *
    RND (1)
40  GOTO 20
```

Modifications that would restrict the kaleidoscope effect to a predetermined area of the screen can be achieved by changing the RND formulas used to select the plot coordinates. For example, to restrict the plotting zone to the lower right quarter of the screen, replace line 30 in the preceding program with

30 PLOT 20*RND (1)+20,20*RND(1)+20

The same RND formula is used in line 30 to generate both coordinates; twenty unique integer values starting with 20 can be generated. The RND formula insures that only points in the lower right quadrant of the low-resolution graphics screen are plotted.

Lines can be placed randomly on the screen as easily as points. Recall that the low-resolution graphics instruction set includes commands for both horizontal and vertical lines. By placing the HLIN and VLIN statements in subroutines, the orientation of the lines can be randomly selected. Variables are assigned random values in the Color Lines program (Listing 2-7), the roles and possible values of the variables are given below.

SS Subroutine Selection for horizontal or vertical line. The set of possible values is $\{1,2\}$.

CO COlor selection for the next line. Values generated for the variable CO are selected from the set of integers $\{0,1,\ldots,15\}$.

FP First Point plotted in the line. The range of possible values is from 0 through 39.

LP Last Point plotted in the line. The range of possible values for LP is from 0 through 39.

AC At Coordinate of the line. AC can take on all integer values in the range 0 through 39.

With just a few minor changes to the Color Lines program, a spectacular patchwork movement can be achieved. The Patches program (Listing 2-8)

Listing 2-7. Color Lines.

```
5  REM   COLOR LINES
10  GR
20 SS = 2 *  RND (1) + 1
30 CO = 16 *  RND (1)
40 FP = 40 *  RND (1)
50 LP = 40 *  RND (1)
60 AC = 40 *  RND (1)
70  COLOR= CO
80  ON SS GOSUB 1000,1100
90  GOTO 20
1000  HLIN FP,LP AT AC: RETURN
1100  VLIN FP,LP AT AC: RETURN
```

displays squares of random size at random areas of the screen in random colors drawn from random directions. The direction (variable D in the program) from which the square is drawn is determined by subtracting the endpoints of the line segment that is plotted. The SGN function returns a value of −1 when its argument is negative, 0 when the argument is 0, and +1 when the argument is positive.

Notice that the AT coordinate variable (AC) takes on all values from the first point (FP) to the last point (LP) of the line. This causes squares to be drawn. The patches displayed when the program runs do not necessarily look like squares. This is because the low-resolution points are longer horizontally than they are vertically.

Listing 2-8. Patches.

```
5  REM   PATCHES
10  GR
20 SS = 2 *  RND (1) + 1
30 CO = 16 *  RND (1)
40 FP = 40 *  RND (1)
50 LP = 40 *  RND (1)
60 D =  SGN (LP - FP)
70  COLOR= CO
75  FOR AC = FP TO LP STEP D
80  ON SS GOSUB 1000,1100
85  NEXT AC
90  GOTO 20
1000  HLIN FP,LP AT AC: RETURN
1100  VLIN FP,LP AT AC: RETURN
```

All of the programs that have created random displays rely upon infinite loops to continue making patterns. Good programming practice, however, dictates that infinite loops not be used. A possible solution to this problem is to place a limit on the number of times the loop is to be executed. Making the following adjustments to the Patches program restricts the number of squares displayed to 100. Add this line:

```
15  FOR X = 1 TO 100
```

Change line 90 to

```
90  NEXT X: END
```

These changes remove the infinite loop.

It may be better, however, to allow the user to determine when the loop should be terminated. Several methods can be used to allow users to do this; two of them will be discussed here. Regardless of the method used, the program should include an instruction which tells the user how to exit the program.

Method 1. Applesoft allows users to interrupt a program by pressing CTRL-C. CTRL-C actually causes an error to occur. Programs can instruct the computer to branch to any line number when an error is encountered by using the ONERR GOTO command. Once the branch is completed, the error instruction must be canceled or subsequent errors cause the same branch to be taken again. Clearing the error branch is accomplished by using the command POKE 216,0. Adding the following lines to the Patches program allows the user to exit the program by pressing CTRL-C.

```
11  HOME:
    PRINT "PRESS <CTRL-C> TO EXIT
    PROGRAM."
```

```
 15  ONERR GOTO 100
100  POKE 216,0
110  END
```

Method 2. A better way of exiting a loop is to include the conditional statement IF-THEN within the loop. We recommend that the condition be based on a user action such as pressing a key. Applesoft allows programs to check if a key has been pressed by looking at the memory location where the most recent keystroke is stored. The statement PEEK (–16384) reads the appropriate memory location; if the value returned by this function is greater than 127, a key has been pressed. Before any significance can be placed on the value returned by this command, any residue from previous keystrokes that happens to be at this location must be removed. The keyboard strobe can be reset with the POKE –16368,0 command. Method 2 can be used in the Patches program by adding the following lines:

```
 15  POKE –16368,0
 90  IF PEEK(–16384) < 128 THEN 20
100  END
```

The programs discussed to this point have allowed the screen coordinates for the graphics displays to be generated within the program. Some applications require the user to specify the coordinates. Two input devices that can facilitate user control of graphing are the keyboard and game paddles. A program that allows a low-resolution picture to be etched by the user is developed in the following paragraphs. The first version uses the keyboard for receiving information from the user. Changes are then made to the program so that it can be used with paddles.

A program should be planned before the actual programming is started. In order to develop a keyboard etching program, consideration should be given to how the user is to determine which points are to be plotted. One alternative is to have the user enter the coordinates of each point. However, such a technique has very limited use. A better approach is to move the plot from point to point based on a series of key presses made at the keyboard. In addition, color changes should be accommodated by the program; the program could allow the user to specify the number that corresponds to the desired color. While the method is suitable, a better choice is to allow the user to change colors by merely pressing a key.

After the input method has been determined, the matter of which keys are to be used for which roles must be decided. A popular scheme used in many games is to use the left arrow to move left, the right arrow to move right, the A key to move up, and the Z key to move down. Many games then allow a bullet or projectile to be released by pressing the space bar. The program being developed uses this scheme, changing the plotting color each time the space bar is pressed. In addition, the user must be able to indicate when the picture is finished. A press of the Q key is adequate for this purpose.

Another question that must be answered is whether a selected move should be continued until the direction is changed by another key press. If the movement is continuous, the user may have difficulty controlling the program. Therefore, the program is written so that a key press is required before each move. A modification presented later allows continuous movement.

The Sketcher program appears in Listing 2-9. The programming instructions that facilitate the moves are discussed in the following paragraphs.

Listing 2-9. Sketcher.

```
5  REM       SKETCHER
10   GR
20 X = 0
30 Y = 0
40 CO = 15
50   COLOR= CO
```

```
60  PLOT X,Y
70  GET M$
80  IF M$ = "A" AND Y > 0 THEN Y = Y - 1
90  IF M$ = "Z" AND Y < 39 THEN Y = Y + 1
100  IF M$ =  CHR$ (21) AND X < 39 THEN X = X + 1
110  IF M$ =  CHR$ (8) AND X > 0 THEN X = X - 1
120  IF M$ = " " THEN CO = CO + 1
130  IF CO > 15 THEN CO = 0
140  IF M$ <  > "Q" THEN 50
150  END
```

Now let's look at the Sketcher program, line-by-line.

Lines 10-40: These statements form an initialization module. Line 10 displays the low-resolution graphics screen. Line 20 sets the initial value of the horizontal coordinate (X) to 0. This initial value insures that the sketch will always start on the left side of the screen. Line 30 sets the initial value of the vertical coordinate (Y) to 0. Since the initial values of X and Y are both zero, sketching will begin in the upper left corner of the screen. Finally, line 40 initializes the color variable (CO) to the value for the color white.

Lines 50-60: The first lines sets the low-resolution color to the value of the variable CO, which is initially 0 (white) because of line 40. Line 60 plots the point on the screen that has coordinates corresponding to the current X and Y variable values.

Lines 70-80: The first line assigns the character corresponding to the key pressed to the string variable M$, and line 80 compares the character represented by M$ to the letter A and checks if the value of Y is greater than 0. If both of the conditions are true, the value of Y is *decremented* (decreased by one), causing the next point plotted to be one position above the previously plotted point. If either comparison is false, the statements following THEN are ignored.

Line 90: This line compares the M$ to the letter Z and checks if Y is less than 39. When both conditions are true, Y is *incremented* (increased by one), and the next point plotted is located one screen position below the previously plotted point. If either comparison is false, the line has no effect on the display.

Line 100: This line checks to see if a left move should and could be made. Since the left arrow character cannot be trapped between quotes like the A in line 80 or the Z in line 90, the comparison must be made in a different manner.

Every character available on the computer has been assigned a unique number called its *ASCII code.* ASCII (pronounced "AS-key") is an acronym for *American Standard Code for Information Interchange.* Any character can be PRINTed or evaluated by using the CHR$ function. The argument for the CHR$ function must be the ASCII code which cooresponds to the needed character. Appendix A contains an ASCII code chart and instructions for its use. The ASCII code for the left arrow character is 21; therefore line 100 compares M$ to the left arrow character.

Line 110: Line 110 makes the appropriate modification to the X variable necessary to move right on the screen. The commands used are similar to those discussed for line 100.

Line 120: This line causes the color variable (CO) to be incremented whenever M$ contains the space character. When typing in line 120, be sure to place the space character between the quotes by pressing the space bar.

Line 130: The low-resolution colors are represented by the numbers 0-15. This line verifies that the value of CO is within that range. If CO exceeds, 15, it is set back to 0.

Lines 140-150: Line 140 instructs the computer to continue plotting points if M$ is not a Q.

Line 150 could be omitted, since it is the last line of the program and the END statement is optional in Applesoft BASIC.

Modifying the program to continue moving in the indicated direction is accomplished by replacing the GET M$ instruction in line 70 with lines 65 and 70 listed below. GET causes execution of the program to stop until a key is pressed; it is therefore necessary to check the keyboard strobe memory location. Inserting the following lines will achieve the desired effect.

```
65  M=PEEK(-16384)
70  IF M >127 THEN M$=CHR$(M-128)
```

Programming the Apple to receive input from game paddles is very easy. The Sketcher program requires fewer instructions when the paddles are used to signal the moves. The paddles are called 0 and 1 by the Apple. The Paddle Sketcher (Listing 2-10) receives the X coordinate from paddle 0 and the Y coordinate from paddle 1. Pressing the button on paddle 0 causes the color variable to be altered, and pressing the button on paddle 1 terminates execution of the program.

The PDL function returns a value from 0 to 255, depending on the position of the paddle dial. The argument for the PDL function is the number corresponding to the paddle from which input is desired. Legal values for the PDL argument range from 0 to 255, but when one set of paddles is used, only 0 and 1 should be referenced. Strange things can happen when other arguments are specified. There is no BASIC command that detects when a paddle button has been pressed. However, if the value returned by PEEK (-16287) is greater than 127, then the button on paddle 0 has been pressed. Similarly, a value greater than 127 returned by the statement PEEK(-16286) indicates that the button on paddle 1 has been pressed.

Listing 2-10. Paddle Sketcher.

```
5  REM       PADDLE SKETCHER
10  GR
20 X =  PDL (0) / 6.5
30 Y =  PDL (1) / 6.5
40  IF  PEEK ( - 16286) > 127 THEN 110
50  IF  PEEK ( - 16287) < 128 THEN 90
60 CO = CO + 1
70  IF CO > 15 THEN CO = 0
80  COLOR= CO
90  PLOT X,Y
100  GOTO 20
110  END
```

Line 65 of the revised Sketcher program assigns the value of the key strobe memory location to the variable M. PEEK (-16384) will return a value greater than 127 whenever a key has been pressed. The new line 70 checks to see if a key has been pressed. If so, the keyed character is assigned to the variable M$. The ASCII code for the character pressed is always 128 less than the value that appears in the key strobe memory location, so the character that corresponds to the pressed key is M-128.

INTEGRATING SOUND WITH LOW-RES GRAPHICS

Programs that cause the computer to make sounds generally have more user appeal than silent programs. The Apple II computer has an internal speaker that can be used to create sound effects. Buzz, click, and beep sounds can be created from Applesoft programs; music, laser, and other sophisticated sounds that must be created with machine language routines are discussed in Chapter 7.

BASIC Programmers can instruct the computer to make a beep sound by using the BELL character, the ASCII code for which is 7. The bell sound can be generated by the statement

```
PRINT CHR$(7)
```

Another way of generating the sound is

```
PRINT "^G"
```

where you have pressed CTRL-G between the opening and closing quotation marks. (When CTRL characters are used within a program, they are invisible.) More than one CTRL-G character can be placed between the quotes, and a bell or beep tone will sound for each bell character placed between the quotes.

The Apple II has a speaker switch that can be toggled by a program by referencing memory location 49200. PEEK(49200) "tweaks" the speaker. One reference to the speaker may not produce an audible sound in some situations; therefore, clicks are achieved by statements like

```
S=PEEK(49200)+PEEK(49200)
```

The more times the speaker switch is referenced in a single statement, the higher the pitch of the sound. If a buzz sound is desired, the statement must be executed several times. The short program in Listing 2-11 generates a buzz sound.

Listing 2-11. Buzzer.

```
5  REM   BUZZER
10 S = 49200
20  FOR X = 1 TO 100
30 Z =  PEEK (S) +  PEEK (S)
40  NEXT X
50  END
```

Generating sounds within a program uses valuable time. When sounds and graphics are included in the same program, care must be taken not to overdo the sound effects or the graphics displays are slowed to an unacceptably slow rate. The Clickity Plot program (Listing 2-12) randomly plots points on the screen. Before each point is plotted, a click is produced.

Listing 2-12. Clickity Plot.

```
5  REM  CLICKITY PLOT
10  GR
15 S = 49200
20  COLOR= 16 *  RND (1)
25 X =  PEEK (S) +  PEEK (S)
30  PLOT 40 *  RND (1),40 *
   RND (1)
40  GOTO 20
```

Inserting the following lines in the above program causes a buzz to be sounded between plots. The time required to make the buzz sound causes the plots to be displayed more slowly than if a click is used.

```
22  FOR Z=1 TO 20
27  NEXT Z
```

APPLICATIONS

The primary purpose of using graphics is to provide illustrations or pictures for supplementing or replacing text. Low-resolution graphics are particularly valuable for creating colorful displays in

game and educational programs. A program that demonstrates how to use low-resolution graphics in a game setting is developed in the following section.

Although the graphics commands are important concepts to master, it is more important to master the techniques essential to sequencing events and formatting displays. In fact, many interesting simulation games have been written that have no graphics displays.

As with any program, a game must be thoroughly planned before the program is written. This does not mean, however, that new ideas concerning game strategy and presentation style will not surface as the game develops. The step-by-step development of a simple game for which low-resolution graphics form a suitable display is now presented.

The first step is to determine the rules for the game. The game selected for this illustration utilizes a game board upon which the players must follow a specific path created by low-resolution graphics. The material presented in this chapter, at this point, provides the background necessary to develop a game of chance for two players. The computer picks a random number for each player's

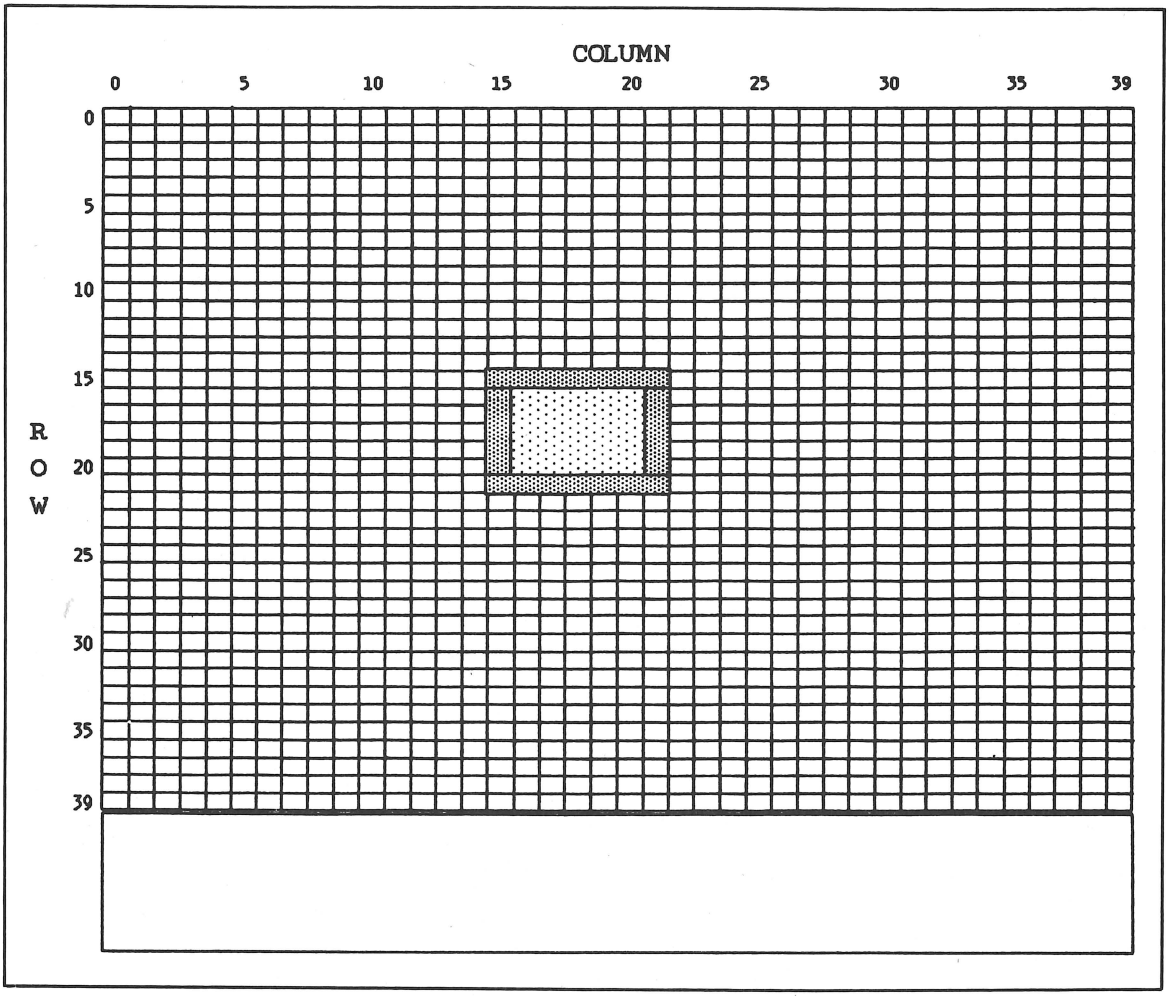


Fig. 2-3. Each square used to create the course for the Pink Out game is five points wide and five points high. The squares are accented with a one point wide border.

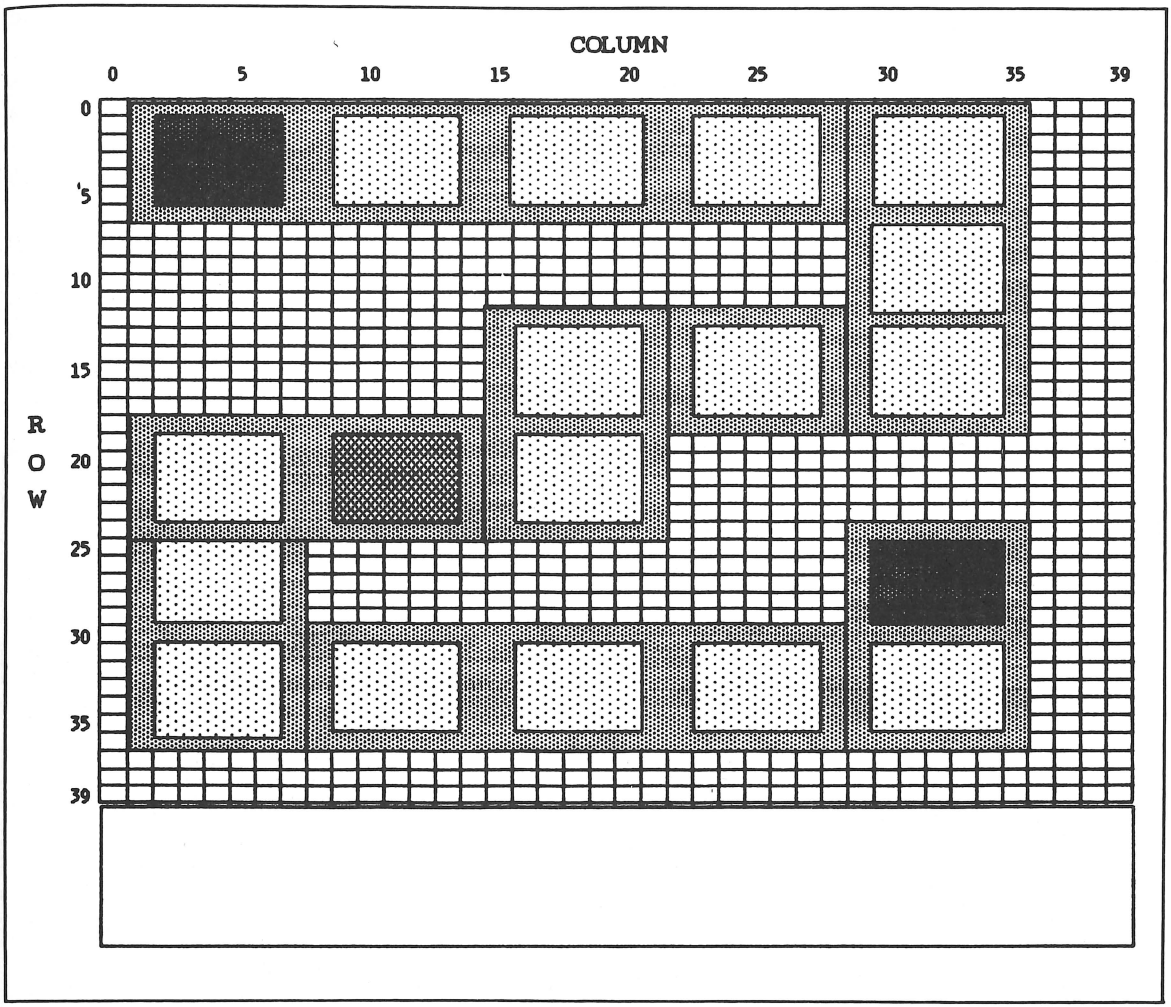


Fig. 2-4. The path used in the Pink Out game should be sketched on graph paper in order to determine the coordinates for placing the squares on the screen.

turn and moves that player's marker the corresponding number of spaces along the path. The first player who lands on the last square of the path wins the game. If the number selected for a player's turn exceeds the number of remaining spaces, the player's marker is not advanced. In order to add an element of suspense to the game, an obstacle is placed along the path. If a player lands on the obstacle, his or her marker is returned to the first square of the path.

Once the game strategy is planned, each detail must be considered. Methods must be developed in low-resolution graphics for displaying the path, markers for the players, and the suspense obstacle. Techniques for generating numbers for a player's turn, sequencing events, and scoring are also needed.

Designing an appropriate path is critical to a game's success. If the path is too long or too short the game quickly becomes boring. While the physical form of the path does not affect the play of the game, an interesting appearance appeals to prospective players. A good way to design the path is to start with the smallest unit in the path, in this

case a square or rectangular shape.

The size of this basic path unit plays an important role in determining the length and appearance of the path. The square should be small enough to allow the path to roam the screen without appearing cluttered, yet large enough to accommodate suitable markers for both players. Since low-resolution graphics points have a large appearance, detailed markers would force the use of very large squares. Experience will provide the judgment necessary to create interesting displays. The basic square used in the demonstration program is seven low-resolution points on a side (see Fig. 2-3), and a backward S-shaped path will be constructed (see Fig. 2-4).

Many styles of markers could be used to represent a player's position on the path, but designs that are not too detailed are most suitable for low-resolution graphics. In this case, both markers must fit on a seven-by-seven matrix. The appearance of the game is enhanced if they can be made even smaller so that the players can easily distinguish their exact positions on the path. The numerals 1 and 2 are used for the demonstration program (see Fig. 2-5). Use of graph paper is extremely helpful in designing the markers.

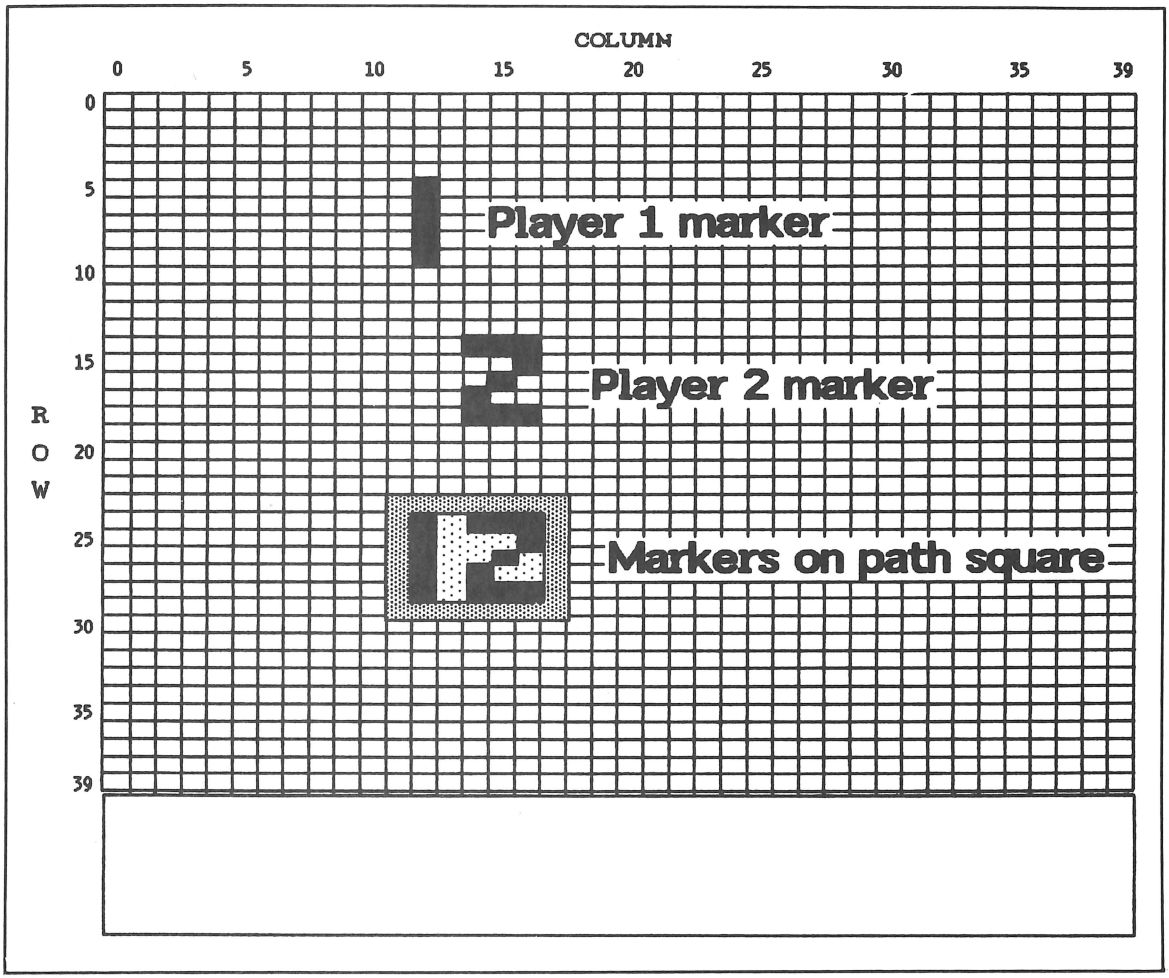


Fig. 2-5. The markers for the players are designed so that both markers can be placed in a single square of the Pink Out game board.

An easily recognized obstacle is best for this type of game. It would be satisfactory to simply use a distinct color for the obstacle square. Naturally, more than one obstacle could be placed along the path; for the sake of clarity, however, only one is used in the demonstration program. The position of the obstacle on the path is randomly selected by the computer each time the game is played.

It is time to consider the colors that are best suited for representing the pieces of the game board. Yellow, blue, and pink were the colors selected for this game. The outer layer of points on all path squares is yellow, the inner points of the first and last squares are dark blue, the obstacle square is pink, and the remaining squares are light blue. If the program is properly written, the colors can be changed very easily.

The game is called Pink Out in honor of the pink obstacle the players must overcome. The program is constructed in a manner that permits it to be altered or extended with minimum programming effort. The coordinates of each square on the path are stored in DATA statements. By changing the information in the DATA statements, new paths can be created. Additional paths could be defined in other DATA statements so that players could choose the design they like best, or a new board could be displayed each time the game is played.

The Pink Out program is printed in Listing 2-13. It is not necessary to include the REM statements; they are simply remarks to help identify the role of different program segments. The program does not branch to any lines with REM statements, so they can be omitted without affecting program execution.

The Pink Out program is written in modular

Listing 2-13. Pink Out game program (continues through page 22).

```
10  REM          * PINK OUT *
20  REM
30  DIM X(25),Y(25)
40  READ NS
50  FOR S = 1 TO NS: READ X(S),Y(S): NEXT S
60  REM
70  REM     *  GAME BEGINS  *
80  REM
90  GR : HOME
100  REM
110  REM     *  DRAW BOARD  *
120  REM
130  CO = 6:FOR S = 2 TO NS - 1: GOSUB 920: NEXT S
140  REM
150  REM * PLOT START AND END SQUARES *
160  REM
170 CO = 2:S = 1: GOSUB 920
180 S = NS: GOSUB 920
190  REM
200  REM * SELECT & SHADE 'PINK' SQUARE *
210  REM
220 P =  INT ( RND (1) * 17 + 2):S = P:CO = 11: GOSUB
     920
230 S = 1:CO = 0: GOSUB 980: GOSUB 1020
240  REM
250  REM  * PICK PLAYER TO START
260  REM
```

```
270 A =  INT ( RND (1) * 2 + 1)
280  REM
290  REM   * SET PLAYER POSITIONS FOR START SQUARE *
300  REM
310 P(1) = 1:P(2) = 1
320  REM
330  REM   * SWITCH PLAYERS *
340  REM
350 A = 3 - A
360  REM
370  REM   * ROLL 4-SIDED DICE *
380  REM
390  HOME : PRINT "PLAYER "A": PRESS ANY KEY TO ROLL D
     ICE.";: GET GO$:R =  INT ( RND (1) * 4 + 1)
400  PRINT : PRINT "YOU ROLLED A "R"."
410  REM
420  REM * CHECK FOR LEGAL ROLL *
430  REM
440  IF P(A) + R > NS THEN  PRINT "SORRY, TOO BIG.": GOTO
     680
450 P(A) = P(A) + R
460  REM
470  REM  * MOVE TO NEW POSITION *
480  REM
490  GOSUB 1090
500  REM
510  REM  * CHECK FOR WIN *
520  REM
530  IF P(A) = NS THEN 730
540  REM
550  REM   * CHECK FOR PINK SQUARE *
560  REM
570  IF P(A) <  > P THEN 680
580  REM
590  REM    * RESTART PLAYER *
600  REM
610  PRINT "YOU PINKED OUT! - RESTART."
620  FOR Z = 1 TO 20:VV = PEEK (49200) + PEEK (49200)
     + PEEK (49200): NEXT Z
630 S = P:CO = 11: ON A GOSUB 980,1020
640 P(A) = 1:S = 1:CO = 0: ON A GOSUB 980,1020
650  REM
660  REM  * DELAY - BRANCH TO NEXT PLAYER *
670  REM
680  FOR Z = 1 TO 1000: NEXT Z
690  GOTO 350
700  REM
710  REM  ****  GAME OVER  ****
720  REM
730  PRINT "YOU WIN!!!"
740  FOR Z = 1 TO 5
```

```
750 CO = 2: GOSUB 1200: ON A GOSUB 980,1020
760 CO = 0: GOSUB 1200:VV =  PEEK (49200) +  PEEK (492
    00): ON A GOSUB 980,1020
770  NEXT Z
780  REM
790  REM  * PLAY ANOTHER GAME? *
800  REM
810  HOME
820  PRINT "PRESS <P> TO PLAY AGAIN OR <Q> TO QUIT.";
830  GET GO$: IF GO$ = "P" THEN 90
840  IF GO$ <  > "Q" THEN 830
850  TEXT : HOME : END
860  REM ========================
870  REM     *  SUBROUTINES  *
880  REM ========================
890  REM
900  REM * PLOT A GAME BOARD SQUARE *
910  REM
920  COLOR= CO: FOR Z = Y(S) TO Y(S) + 4: HLIN X(S),X(
    S) + 5 AT Z: NEXT Z
930  COLOR= 15: VLIN Y(S) - 1,Y(S) + 5 AT X(S) + 6: VLIN
    Y(S) - 1,Y(S) + 5 AT X(S) - 1
940  HLIN X(S),X(S) + 5 AT Y(S) - 1: HLIN X(S),X(S) +
    5 AT Y(S) + 5: RETURN
950  REM
960  REM     * PLAYER 1 MARKER *
970  REM
980  COLOR= CO: VLIN Y(S),Y(S) + 4 AT X(S) + 1: RETURN
990  REM
1000  REM     * PLAYER 2 MARKER *
1010  REM
1020  COLOR= CO: HLIN X(S) + 3,X(S) + 5 AT Y(S)
1030  PLOT X(S) + 5,Y(S) + 1: PLOT X(S) + 4,Y(S) + 2
1040  PLOT X(S) + 3,Y(S) + 3: HLIN X(S) + 3,X(S) + 5 AT
    Y(S) + 4
1050  RETURN
1060  REM
1070  REM    * MOVE PLAYER'S MARKER *
1080  REM
1090  FOR S = P(A) - R TO P(A) - 1
1100 CO = 0: ON A GOSUB 980,1020
1110 CO = 2 + (S > 1) * 4 + (S = P) * 5
1120  GOSUB 1200: ON A GOSUB 980,1020
1130 VV =  PEEK (49200) +  PEEK (49200)
1140  NEXT S
1150 CO = 0: ON A GOSUB 980,1020
1160  RETURN
1170  REM
1180  REM      * TIME DELAY *
1190  REM
```

```
1200  FOR VV = 1 TO 200: NEXT VV: RETURN
1210  REM ======================
1220  REM  COORDINATES OF SQUARES
1230  REM       FOR GAME BOARD
1240  REM ======================
1250  DATA 19,1,1,8,1,15,1,22,1,29,1
1260  DATA 29,7,29,13,22,13,15,13,15,19
1270  DATA  8,19,1,19,1,25,1,31,8,31,15,31,22,31,29,31
     ,29,25
```

form, with each series of lines having a particular set of functions to perform. Module One is the initialization module, extending from line 10 through line 310. Instructions in Module One are executed before the actual play of the game begins.

Line 30: Arrays X and Y are used to store the coordinates for placing the squares on the path. X(n) represents the horizontal coordinate of the Nth square on the path, and Y(n) represents the corresponding vertical coordinate. Due to the size of the individual squares, it is not advisable to design paths that are longer than 25 squares. Dimension-arrays to the maximum value desired allows the paths to be altered or extended without having to change line 30.

Line 40: NS is the variable for the Number of Squares on the path. The value assigned to NS is stored in the DATA statement in line 1250. The program can be modified to display longer or shorter paths by changing the first datum in line 1250.

Line 50: The horizontal and vertical coordinates of the squares that form the path are placed in the X and Y arrays. S is used throughout the program to reference squares along the path.

Line 90: Before the board can be placed on the screen, the low-resolution graphics mode must be set with the command GR. HOME erases any textual information that appears on the four lines reserved for text at the bottom of the screen.

Line 130: This line forms a loop that repeatedly transfers execution to a subroutine (line 920) that places a square on the path. Values that must be passed to the subroutine are the position of the Square in the path (S) and the COlor (CO) that is to be used for the square. This line uses the subroutine to place light blue squares on the path. The first and last squares are omitted from the loop, since they are to be dark blue.

Lines 170-180: These lines place the first and last squares of the path on the screen with a dark blue color.

Line 220: This line randomly selects the position of the pink obstacle square on the path. The values for placing the pink square range from 2 to one less than the number of squares on the path (NS–1). The number of squares in this version of Pink Out is 19. This means that 17 squares are candidates to be colored pink. The first permissible pink square is in position 2 on the path, so 2 is the initial value of the range. The pink position (P) is assigned to the Square variable (S), the color is set to pink, and the square is placed on the screen when subroutine 920 is executed again.

Line 230: This line causes the markers for both players to be placed on the first square of the path. The markers will be displayed in black. The marker for player 1 is displayed by subroutine 980, and subroutine 1020 places the marker for player 2.

Line 270: The variable A will be used to keep track of the players' turns. The value of A will be 1 when it is player 1's turn and 2 when it is player 2's turn. Line 270 randomly selects which player will go first by assigning either 1 or 2 to this variable.

Line 310: Each player's position on the path must be known at all times. The P array is used to perform this task. Player 1's current square position is stored in P(1) and player 2's position is stored in P(2). Line 310 initializes both values to 1, since both players will start in square 1 of the path.

Lines 350 through 850 form Module Two. Player instructions, sequencing game events, and

scoring are accomplished in this segment of the program. Each time the module is executed, the players' turns are switched. Execution of the module continues until one of the players lands on the last square of the path. Each line of the module is discussed below.

Line 350: The value of the variable A alternates between 1 and 2 each time line 350 is executed. A has the value 1 when it is player 1's turn and 2 when it is player 2's turn. Alternating the value of A between the values 1 and 2 allows one carefully written module to be used to simulate either player's turn.

Lines 390-400: Players are instructed to press any key to simulate a roll of dice when it is their turn to move. The key press obtained is assigned to the variable GO$, but the particular key pressed does not affect the roll of the dice. The value of the roll is restricted from 1 to 4 since the board is small, and the randomly generated value is assigned to the variable R. Players are informed of the number rolled by line 400.

Line 440: Line 440 checks whether the roll would cause the player's marker to be moved beyond the last square on the path, position NS. P(A) is the current player's position on the path; P(A)+R is the new position of the marker if the move can be made. If the move is not possible, then the player's marker is not moved, he or she is told that the roll is too big, and the next player is allowed to take a turn.

Lines 450-490: The value of the roll, R, is added to the current player's path position variable P(A), and subroutine 1090 moves a player's marker to the new position. Line 490 simply causes the subroutine to be executed. The subroutine is explained in the Module Three discussion.

Line 530: The current player's path position, P(A), is compared to the number of positions on the path, NS. If these values are equal then the player has landed on the last square and the program branches to line 730, where the player is declared the winner of the game. If the values are different, line 570 is the next significant line executed.

Line 570: A check to determine if the player landed on the pink obstacle square is performed in line 570. If the player's position P(A) is not equal to the obstacle's Path position (P), then a branch is made to end the current player's turn at line 680. If the positions are the same, then lines 610 through 640 are executed.

Line 610: The player is informed that the pink obstacle has been encountered. The phrase *pink out* is used to describe the event, and the player is instructed to move back to the first square of the path.

Line 620: An audio signal (a buzz) sounds to alert the player that the obstacle has been encountered. The buzz may be made longer or shorter by changing the loop limit parameter (20). Recall that the pitch or tone of the buzz can be modified by altering the number of references made to the speaker switch, PEEK (49200).

Line 630: Since the player's marker is to be moved back to the first square on the path, the marker must be removed from the pink square. This is accomplished in line 630 by setting the square pointer (S) to the pink square (P) and the color (CO) to pink before redrawing the appropriate marker. The marker appears to vanish from the pink square, since it is also pink after the subroutines at line 980 or 1020 are executed.

The ON-GOSUB statement is used to determine which subroutine should be used. When the value of A is 1, the subroutine that appears in the first position after the word GOSUB is executed; when the value of A is 2, the subroutine named in the second position is executed. Although those are the only possible values of A in this program, it is worth noting that if A has a value greater or less than the number of branches listed after the word GOSUB, program execution continues with the next BASIC statement after the ON-GOSUB. The legal values for an ON parameter are 0 through 255. An ILLEGAL QUANTITY error is displayed if the value of the parameter is not in the stated range.

Line 640: The player path position variable P(A) is reset to 1, since the obstacle was encountered. The players marker is repositioned in the first square of the path using the color black. Again, the ON-GOSUB structure is used to determine which marker is to be displayed.

Lines 680-690: A time delay is executed at the end of each player's turn, so that the comments displayed by the program may be read. Depending on the age or reading rate of the intended player, the time delay may need to be longer or shorter. Line 690 causes a branch back to the beginning of Module Two for the next player's turn.

Lines 730-740: This line is executed only if a player's marker lands on the last square of the path. If that event occurs, the player is declared the winner. The purpose of the loop set up in line 740 is to reward the winner by flashing the player's marker in the finish square. The loop is executed five times. If more flashes or blinks is desired, change the loop limit parameter (5).

Lines 750-770: Flashing is accomplished by alternating the color used for displaying a marker. Line 750 displays the winner's marker in the dark blue color of the last square. The short time delay subroutine (line 1200) is executed to slow the flash speed. Line 760 places the winning player's marker back in the finish square using the color black. Again the time delay in subroutine 1200 is needed to slow down the appearance of the flash. A click sound is generated immediately before displaying the marker. Finally, line 770 simply completes the flash sequence loop, ending the game.

Lines 820-840: After the text lines are cleared, the players are prompted to press the P key to play again or the Q key to quit. The character of the key pressed is assigned to the string variable GO$. The program then compares GO$ to the letter P. If the key pressed was P, the program branches to line 90 to start another game. If additional, different boards are desired, the branch could be made to line 40 to READ the information about the next path before a new game is started. Line 840 forces the players to press the Q key if they want to quit, preventing the program from ending in the event of an accidental keystroke that was not a P or Q.

Line 850: This is the logical end of the program. The text mode is reinstated and the screen is cleared before the END statement. It is a good idea to display the text screen before ending any program that uses graphics. If another program is run while a graphics screen is displayed, it may alter the appearance of the program and confuse the user.

Module Three of the Pink Out program contains the subroutines needed for displaying the paths, and displaying and moving the players' markers.

Line 920: The inner points of a square are displayed using the color assigned to the variable CO. S is the variable that indicates the relative position on the path of the square being displayed. X(S) is the key horizontal coordinate for the square, while Y(S) is the key vertical coordinate. The inner portion of each square is five points vertically by six points horizontally. The width of the extra horizontal point provides the added room necessary to fit two markers into the same square.

Lines 930-940: Line 930 displays the vertical lines for the yellow border of a path square while the yellow horizontal lines of a square's borders are displayed in line 940.

Line 980: Player 1's marker is placed in the left half of the inner portion of a square. The color used for displaying the numeral 1 is determined by the value of the variable CO.

Lines 1020-1050: The series of statements in these lines places the marker for player 2 in the right half of the inner portion of the path square designated by the variable S.

Lines 1090 through 1160 are responsible for moving the markers of the players as they proceed along the game path. The marker is temporarily displayed in each square from its current position to its new position.

Line 1090: This line sets the initial and limit values for the FOR-NEXT loop. The limit is set at one less than the new position. Since the marker will also be removed from the square within the loop, the final placement of the marker will take place beyond the loop.

Line 1100: This line sets the color variable for black and calls the subroutine for the appropriate marker.

Line 1110: Once the marker has been placed in a square it must be removed before displaying it in the next square of the path. The difficulty in accomplishing this task is that not all the squares are the same color. Some special logic is used in

line 1110 to determine the color of the square in which the marker is situated. The relationship between the color of the square and the position of the square on the path is used to determine the color that is needed to make the marker disappear. The expression that computes that value that is assigned to the variable CO in line 1110 may be new to you. The statement has the same outcome as the following sequence of statements:

```
1 CO=2
2 IF S>1 THEN CO=CO+4
3 IF S=P THEN CO=CO+5
```

In order to fully understand the expression in line 1110, an explanation of the above sequence of lines is necessary. A value of 2, for dark blue, is assigned to CO in line 1. This sets the color variable to the value needed for the first and last squares of the path. Line 2 adds 4 to the color variable if other than the first square of the path is indicated by the variable S. The fact that the last and first squares are both dark blue is not relevant, since the marker would not be removed if it appeared in the last square of the path. If the condition S>1 in line 2 is true, then the value of CO becomes 6, the color code for light blue. Line 3 adds 5 to the value of CO if the referenced square is the pink obstacle. All the logic of lines 1, 2, and 3 is combined in line 1110 of the program.

Any condition in an IF-THEN statement has a truth value of either 1 or 0. If the condition is true, the truth value is 1; if not, its truth value is 0. The fact that a condition has a truth value allows conditional expressions in places other than IF-THEN statements. As a matter of fact, conditions can be placed in any mathematical expression, and their truth values can be evaluated as part of the expression.

In line 1110, CO will always have at least a value of 2. The truth value of the condition S>1 is then multiplied by 4, and the truth value of the condition S=P is multiplied by 5. Remember that the product of any number and 0 is 0 and that the product of any number and 1 is that number. If S>1 has a value of 1, then 1*4 is 4, so 4 would be added to total value to be assigned to CO. However, if S>1 has a value of 0, then 0*4 is 0, and would not add value to CO. Similarly, S=P is evaluated as 1 or 0 and multiplied by 5. After the expression is evaluated and assigned to CO, CO will have a value of 2, 6, or 11 depending upon the color of the square containing the marker.

Lines 1120-1130: A brief time delay subroutine in line 1200 is executed to allow the players to observe the placement of the marker in the square. Then the marker is removed by displaying it using the color determined in line 1110. A click is sounded by line 1130 to call attention to the placement of the marker.

Lines 1140-1150: This line completes the loop for moving the markers, and line 1150 places the marker in the final square of the current move. The value of S is changed by line 1140, so that its value by the time 1150 is executed is the player's new position on the board.

Line 1160: Control returns to the statement following the GOSUB that called the subroutine.

Line 1200: A time delay subroutine appears in line 1200. Time delays simply keep the computer busy counting to itself, so that execution of subsequent statements is delayed temporarily.

Lines 1250 through 1270 form the Data Module of the program. The first datum is the number of squares that are to be used to form the game path. The remaining data are the key coordinates of the reference points used for placing the squares on the path. The coordinates are entered in the data statements as ordered pairs in the horizontal coordinate-vertical coordinate form. All the information contained in Module Four could be placed in one DATA statement if desired.

The Pink Out game could be converted to an educational game program rather than a simple chance game. A series of questions could be asked of the players before they are allowed to take their turns. If the players' responses to a question are incorrect they lose their next turn. Any subject area could be introduced into the program. The players could be asked to solve math problems, name presidents, name the capitals of states, or identify symbols of elements, just to mention a few possibilities.

3

High-Resolution Graphics

Graphics applications that require detailed illustrations or pictures cannot be accomplished using low-resolution graphics. Low-resolution plots are too large and are not available in enough screen positions. For finer detail, the high-resolution graphics mode should be used. In this chapter we will discuss the high-resolution graphics instruction set, the properties of high-resolution graphics, high-resolution graphics screen management, graphics applications, and "page flipping."

HIGH-RESOLUTION INSTRUCTION SET

The commands that are used to create high-resolution displays are similar to those used for low-resolution graphics. The Applesoft commands used for high-resolution graphics are introduced and explained in this section.

HGR. HGR instructs the computer to clear and display high-resolution page 1. Since the memory segment used for page 1 is erased when HGR is used, the command must be executed before plotting high-resolution points. Four lines at the bottom of the screen may be used for textual information when page 1 is displayed.

HGR2. Page 2 of high-resolution graphics is cleared and displayed when this command is executed. No part of the screen is allocated for text when HGR2 is used.

HCOLOR. Before plotting a point, the desired color must be specified. The proper form for the HCOLOR command is

HCOLOR = {n}

where n is a number corresponding to the desired color. For example, HCOLOR=3 causes plotted points to appear white.

HPLOT. Points on the selected high-resolution page are displayed using the HPLOT command. The correct syntax for the statement is

HPLOT {x,y}

where x is a number from 0 through 279 corresponding to the horizontal screen position, and y is a number from 0 through 191 corresponding to the ver-

tical screen position of the point to be plotted. The statement HPLOT 0,0 causes a point to be plotted in the upper left corner of the screen. When page 1 is used for displaying high-resolution graphics, points with vertical positions from 160 to 191 do not appear on the screen. An ILLEGAL QUANTITY error occurs if x is less than 0 or greater than 279, or if y is less than 0 or greater than 191.

HPLOT-TO. No high-resolution commands are available specifically for plotting horizontal and vertical lines. Instead, the HPLOT-TO command can be used for drawing horizontal, vertical or slanted lines. The proper syntax is

HPLOT {x1,y1} TO {x2,y2}

where x1 and x2 are the horizontal coordinates and y1 and y2 are the vertical coordinates of the endpoints of the desired line segment. Several adjoining line segments may be displayed by extending the statement with additional TO parameters. For example

HPLOT 0,0 TO 279,0 TO 140,159 TO 0,0

causes a large triangle to be displayed on the screen. Since each program line can contain only approximately 239 characters, it may sometimes be necessary to continue an HPLOT-TO statement on another line. When a line segment is to be extended from a previously plotted point the x1,y1 parameters may be omitted from the statement. The above triangle can be enclosed in a rectangle by using the following command:

HPLOT TO 0,159 TO 279,159 TO 290,0

No lines appear if this statement is not preceded by either an HPLOT or an HPLOT-TO command.

TEXT. TEXT is also used to exit the high-resolution graphics mode. When the text screen is displayed, any information that was on the text screen before the use of HGR or HGR2 reappears.

PROPERTIES OF HIGH-RESOLUTION GRAPHICS

High-resolution graphics allows up to 53,760 unique points on the screen to be plotted. Where each low-resolution graphics point is a 7 × 4 matrix of pixels, each point in high-resolution graphics is one pixel.

Two separate memory segments are reserved for high-resolution graphics. Either segment can be displayed on the monitor screen to form high-resolution displays. These two memory segments are called high-resolution *pages* 1 and 2. Both pages have 280 horizontal positions for plotting points. Page 1 can normally display plots at 160 vertical positions, and page 2 has 192 vertical positions. The difference in vertical positions arises from the fact that four lines of text can be displayed below the page 1 screen, while page 2 uses the entire screen.

Although more finely detailed displays can be achieved using high-resolution graphics, the colors available are significantly reduced. Only eight colors are available in high-resolution graphics mode; these are listed in Table 3-1. Notice that two of the colors are black and two are white, apparently reducing the number of unique colors available to six. Furthermore, not all colors can be displayed at every screen position. Odd-numbered colors can be displayed in odd-numbered columns (horizontal positions), and even-numbered colors can be displayed only in even-numbered columns. Exceptions to the rule are the white and black colors, which are visible in all columns.

It is helpful to become familiar with the color restrictions before attempting to develop programs which use high-resolution graphics. Most of the problems arise when either single points or vertical

Table 3-1. High-Resolution Colors.

Code	Color	Code	Color
0	black (1)	4	black (2)
1	green	5	orange
2	violet	6	blue
3	white (1)	7	white (2)

lines are plotted. Enter the following line in immediate mode:

```
HGR: HCOLOR = 2: HPLOT 0,0
```

After the above line is executed, a single high-resolution point is displayed in the upper left corner of the screen, violet in color if the TV or monitor is properly adjusted. The point can be extended to a vertical line by the following statement:

```
HPLOT TO 0,159
```

The fact that odd-numbered colors other than 3 and 7 cannot be displayed in even-numbered columns can be confirmed by entering the following statement:

```
HCOLOR=1: HPLOT 0,0 TO 0,159
```

The violet line disappeared when the statement was executed, and the expected green line was never displayed. Entering the following statement causes an orange line to be displayed:

```
HCOLOR=5: HPLOT 1,0 TO 1,159
```

The white colors 3 and 7 can be displayed in any column, but the points do not necessarily appear to be white. The color will depend on which white color code is being used and whether the point is displayed in an even or odd column of the screen. Many difficulties associated with high-resolution graphics colors can be overcome by using the white colors. Experimenting with different colors in different columns of the screen is a good way to become familiar with the color restrictions.

Horizontal and vertical lines formed with the HPLOT-TO command are sharp and straight. Some oblique or slanted lines, however, have a jagged appearance. Points are not always available along a geometrically straight line between two points on the screen, so the computer lights points that are closest to the ones needed. Enter the following statements in immediate mode:

```
HGR: HCOLOR=3: HPLOT 1,1 TO 200,5
```

Notice that the displayed figure looks more like a staircase than a straight line.

This fact, in conjunction with the unusual color properties of high-resolution graphics, can be frustrating at times. Sometimes the displays obtained are unexpected, but are very beautiful. The key to creating interesting high-resolution graphics displays is being familiar enough with the plotting and color properties so that they can be regarded as enhancements rather than limitations.

The Color Sweep program (Listing 3-1) demonstrates how interesting designs due to the color properties mentioned above can be created with high-resolution graphics.

Listing 3-1. Color Sweep.

```
10  REM      COLOR SWEEP
20  HGR
30  HCOLOR= 2
40 A = 140:B = 80:V = 2
50  FOR X = 0 TO 279 STEP V
60  HPLOT A,B TO X,0
70  NEXT
80  FOR Y = 0 TO 159 STEP V
90  HPLOT A,B TO 279,Y
100  NEXT
110  FOR X = 278 TO 0 STEP  - V
120  HPLOT A,B TO X,159
130  NEXT
140  FOR Y = 159 TO 0 STEP  - V
150  HPLOT A,B TO 0,Y
160  NEXT
```

The Color Sweep program draws lines from the center of the high-resolution graphics screen to points on the edge using the HPLOT-TO command. Line segments are placed on the screen in a clockwise order, starting at the upper left corner of the screen.

Lines 20-30: HGR displays and clears high-resolution page 1, and line 30 sets the color for drawing the lines to violet (2).

Line 40: A is the variable used to name the horizontal coordinate of the center of the screen, 140. B is the variable used to store the value of the vertical coordinate of the center of the screen, 80. The point at the center of the screen serves as an endpoint for each line segment drawn in the program. V is used for the step parameter in the loops that draw the line segments; it is initialized to 2 so that line segments connect the point at the center of the screen with every other point along the screen border.

Line 50: The parameters of the loop that draws line segments from the center of the screen to points along the top of the screen are set in line 50. The loop variable X takes on the even number values from 0 to 279. X serves as the horizontal coordinate for endpoints of the line segments drawn in the loop in lines 50 through 70 of the program.

Lines 60-70: Line 60 draws the line segments from the center (A,B) to points along the top of the screen (X,O) as the loop variable X assumes the values discussed for line 50. Since the vertical coordinate of all points along the top of the screen is 0, the constant 0 is used in the statement. Line 70 terminates the loop that draws the line segments described in lines 50 and 60.

Lines 80-100: The loop formed by these lines draws line segments from the center point of the screen to the points along the right edge of the screen. The variable Y serves as the vertical coordinate for the points on the right edge of the screen and takes on even values from 0 to 159. The horizontal coordinate of any point on the right edge of the screen is 279, so the constant 279 is used.

Lines 110-130: Drawing line segments from the center of the screen to points along the bottom of the screen is accomplished in this loop. STEP–V is used so that the segments are placed on the screen in a clockwise order. Variable roles are similar to those in lines 50-70. The constant 159 is used for the vertical coordinate of all points along the bottom of the high-resolution page 1.

Lines 140-160: The design is completed by this loop, which draws line segments from the center to points along the left edge of the screen. As was the case in line 110, –V is used as the STEP parameter to place the lines on the screen in clockwise order. This program segment is similar to lines 80 through 100.

By changing the HCOLOR parameter and the V value, Color Sweep can be made to display several variations of its design. If the program is allowed to run continuously, selecting the colors at random, then it becomes an interesting work of computer art. The modification can be made by changing line 30 to

```
HCOLOR=RND(1)*8
```

and adding the following line:

```
170  GOTO 30
```

A modification that can provide even more interesting visual effects is to make the changes:

```
40  A=RND(1)*280:
    B=RND(1)*160:
    V=RND(1)*10+1
```

when these changes are implemented, the line segments are no longer drawn from the center of the screen, but rather from a point randomly selected by the computer. The design changes drastically with each pass through lines 30 to 170, since the STEP parameter is also selected at random by the program.

SCREEN MANAGEMENT

One aspect of high-resolution graphics programming that differs significantly from low-resolution programming is the number of screen points. Having more screen points to manage sometimes makes the programming very tedious, but the programming effort may often be reduced by applying a variety of mathematical methods. Several useful programming techniques that utilize varying degrees of mathematical complexity will be discussed throughout this chapter. A strong math background is needed to understand how many of

the program segments work; most of the techniques may be utilized in a program, however, without completely understanding the underlying mathematical concepts.

Polygons are useful figures for creating graphics displays. Examples of polygons are triangles, squares, rectangles, pentagons, and octagons. The common property of all these figures that allows them to be described by the one word *polygon* is that they consist of straight line segments enclosing an area. The word itself has two syllables, *poly* meaning "many," and *gon* meaning "sides." Polygons then, are many-sided figures that can represent a wide variety of the objects encountered in our environment.

Perhaps the most useful polygon for creating graphics images is the *rectangle*. Rectangles have four sides and four square (90-degree) *vertices* (corners). Opposite sides of rectangles are the same length. The measurements of the sides are commonly called the length and width of the rectangle. A rectangle can be displayed on the monitor with high-resolution graphics by using the HPLOT-TO command to form line segments that connect four points on the screen. The program in Listing 3-2 displays a rectangle that forms a border around the page 1 high-resolution graphics screen.

Important observations should be made about line 30. Notice that the first and last coordinate pairs are the same, consecutive points in the statement have either the same horizontal coordinate or the same vertical coordinate, neither coordinate is shared by three consecutive points, and only two unique horizontal coordinates and two unique vertical coordinates are used.

General Rectangle, another program that forms the same rectangle, appears in Listing 3-3. It assigns the horizontal coordinate of the upper left corner to the variable A, the vertical coordinate of that point to the variable B, the length (horizontal measurement) of the rectangle to the variable L, and the width (vertical measurement) of the rectangle to the variable W. All the other points used to construct the rectangle are determined using the variables A, B, L, W, and the geometric properties of rectangles.

Listing 3-2. Rectangle.

```
5  REM   RECTANGLE
10  HGR
20  HCOLOR= 3
30  HPLOT 0,0 TO 279,0 TO 279,159 TO 0,159 TO 0,0
40  END
```

Listing 3-3. General Rectangle program.

```
10  REM    GENERAL RECTANGLE
20  HGR
30  HCOLOR= 3
40 A = 0
50 B = 0
60 L = 279
70 W = 159
80  HPLOT A,B TO A + L,B TO A + L,B + W TO A,B + W TO
     A,B
90  END
```

Listing 3-4. T-Form program.

```
5  REM     T FORM
10  HGR
20  HCOLOR= 3
30 A = 100:B = 50:L = 50:W = 10
40  GOSUB 1000
50 A = 120:B = 60:L = 10:W = 40
60  GOSUB 1000
999  END
1000  HPLOT A,B TO A + L,B TO A + L,B + W TO A,B + W TO
      A,B
1010  RETURN
```

The observation made regarding the Rectangle program applies to General Rectangle as well. While Rectangle is the short program of the two, General Rectangle is more easily modified for displaying different rectangles. Change line 60 to 60 L = 10 and run the program again. A very different appearing rectangle is displayed as the result of the minor modification. Experiment with the General Rectangle program by altering the values of each variable. The point specified by the coordinates A,B is always the upper left vertex of the rectangle. The value of L must be less than 280-A, and the value of W must always be less than 160-B. For example, if A is assigned a value of 100 and B is assigned a value of 20, then L must have a value less than 180 and W must have a value less than 140.

Should it be necessary to use more than one rectangle to create a display, the statement in line 80 of General Rectangle could be used in a subroutine. The values for A, B, L, and W need to be specified before the subroutine is called. T-Form is a program that illustrates this technique (Listing 3-4).

The T-Form program displays a T-shaped figure constructed with two rectangles. The program was presented in its simplest form, but a number of improvements suggest themselves. A better program design is to choose a reference

Listing 3-5. Scalar T.

```
5  REM      SCALAR T
10  HGR
20  HCOLOR= 3
30 KH = 100
40 KV = 50
50 S = 1
60 A = KH:B = KV:L = 50 * S:W = 10 * S
70  GOSUB 1000
80 A = KH + 20 * S:B = KV + 10 * S:L = 10 * S:W = 40 *
     S
90  GOSUB 1000
999  END
1000  HPLOT A,B TO A + L,B TO A + L,B + W TO A,B + W TO
      A,B
1010  RETURN
```

point on the figure and specify the values of A and B for each of the rectangles with expressions that describe their positions relative to the reference point. Such a modification allows the T-shape to be placed at any location on the screen merely by changing the coordinates for the reference point. Another change that makes the program more usable is to provide for a *scaling variable* that allows the size of the figure to be altered easily. The Scalar T program (Listing 3-5) applies both of these con-

Listing 3-6. Robot plotting program.

```
5  REM           ROBOT
10  HGR
20  HCOLOR= 3
30 S = 1
40 KH = 50:KV = 0
50  HPLOT KH,KV TO KH + 6 * S,KV + 7 * S: REM   LEFT A
    NTENNA
60  HPLOT KH + 20 * S,KV TO KH + 14 * S,KV + 7 * S: REM
     RIGHT ANTENNA
70 A = KH:B = KV + 8 * S:L = 20 * S:W = 18 * S: GOSUB
    1000: REM   HEAD
80 A = KH + 4 * S:B = KV + 13 * S:L = S * 2:W = S * 3:
     GOSUB 1000: REM   LEFT EYE
90 A = KH + 14 * S: GOSUB 1000: REM  RIGHT EYE
100 A = KH + S * 4:B = KV + S * 20:L = S * 12: GOSUB 1
    000: REM   MOUTH
110 A = KH + 6 * S:B = KV + 26 * S:W = S * 5:L = S * 8
    : GOSUB 1000: REM   NECK
120 A = KH - 6 * S:B = KV + 31 * S:L = S * 32:W = S *
    22: GOSUB 1000: REM   UPPER BODY
130 A = KH - 16 * S:L = S * 10:B = KV + 33 * S:W = 15 *
    S: GOSUB 1000: REM   LEFT ARM
140 A = KH - 14 * S:L = S * 6:W = S * 3:B = KV + 48 *
    S: GOSUB 1000: REM    LEFT HAND
150 A = KH + 26 * S:L = S * 10:B = KV + 33 * S:W = 15 *
    S: GOSUB 1000: REM   RIGHT ARM
160 A = KH + 28 * S:L = S * 6:W = S * 3:B = KV + 48 *
    S: GOSUB 1000: REM   RIGHT HAND
170 B = KV + 53 * S:W = S * 5:L = S * 12:A = KH + 4 *
    S: GOSUB 1000: REM   MID SECTION
180 A = KH - 10 * S:B = KV + 58 * S:L = S * 40:W = 15 *
    S: GOSUB 1000: REM   LOWER BODY
190 A = KH - 4 * S:B = KV + 73 * S:W = S * 10:L = S *
    10: GOSUB 1000: REM    LEFT LEG
200 A = KH + 14 * S:B = KV + 73 * S:W = S * 10:L = S *
    10: GOSUB 1000: REM   RIGHT LEG
999  END
1000  HPLOT A,B TO A + L,B TO A + L,B + W TO A,B + W TO
    A,B
1010  RETURN
```

cepts. The variable KH stores the horizontal coordinate of the reference point, KV the vertical coordinate of the reference point, and S the scaling factor for the size of the T-shape.

By changing the values of H and K in the Scalar T program, the figure can be placed at any location on the screen. The value of S can be increased to display a bigger figure or decreased to display a smaller figure. To observe the figure using several different scaling values, make the following changes to the Scalar T program. Add these lines:

```
  8  FOR S=.5 TO 2 STEP .1
100  GET G$
110  NEXT S
```

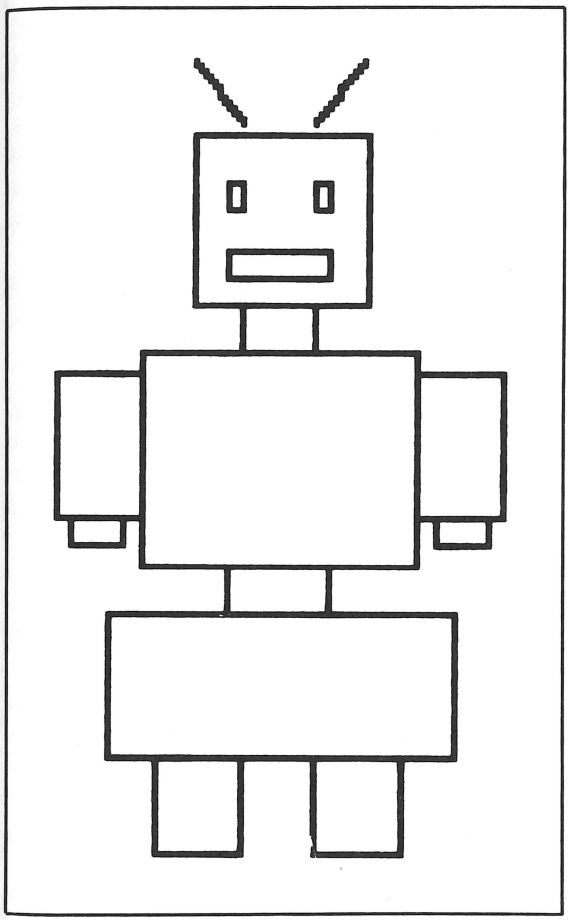


Fig. 3-1. This high-resolution display is created by the Robot program.

and omit line 50. Inserting line 100 prevents the program from displaying the figures so quickly that they cannot be observed. A keystroke must occur between successive displays.

The program in Listing 3-6 applies the scaling technique to a more complex figure, the robot shown in Fig. 3-1. Lines 50 and 60 in the program form the robot's antennas, which are the only parts that are not rectangles. The roles of the variables A, B, KH, KV, L, W, and S are the same as in the Scalar T program. Remarks stating the robot part described by each line are included.

All of the preceding programs display outlines of rectangles to form the desired figure using the HPLOT-TO command. Solid rectangles can be formed by displaying lines that connect corresponding points on opposite sides of the rectangles. Either horizontal or vertical lines may be used, but horizontal lines are generally preferred due to the color properties of high-resolution graphics. A single solid rectangle is displayed by the program in Listing 3-7. The roles of A, B, L, and W remain unchanged from the General Rectangle program. VC is the loop variable used to indicate the Vertical Coordinates of the endpoints of the horizontal lines needed to shade the rectangle.

Listing 3-7. Solid Rectangle.

```
5  REM     SOLID RECTANGLE
10  HGR
20  HCOLOR= 3
30 A = 120:B=50:L=50:W=10
40  FOR VC = B TO B + W
50  HPLOT A,VC TO A + L,VC
60  NEXT VC
70  END
```

The Scalar T program can display a solid T figure by replacing lines 1000 and 1010 with the lines listed below.

```
1000  FOR VC=B TO B+W
1010  HPLOT A,VC TO A+L,VC
1020  NEXT VC
1030  RETURN
```

This same subroutine (1000-1030) can be placed in the Robot program to cause it to display a solid-colored robot. Without additional modifications the eyes and mouth of the robot cannot be seen, since they are the same color as the head. The color should be set using the HCOLOR command before calling the rectangle subroutine if the rectangle is to be displayed using a different color. Inserting the following lines in the modified Robot program will cause the robot to be displayed in white and the eyes and mouth in black.

```
75   HCOLOR=0
105  HCOLOR=3
```

Programming strategies similar to those used to display rectangles can be used to create triangles, pentagons, and other polygons. This technique is straightforward in a mathematical sense and is suitable for many applications.

Hourglass Timer (Listing 3-8) is a program that uses the techniques discussed up to this point in the chapter to create the illusion of sand sifting

Listing 3-8. Hourglass Timer.

```
10  REM      HOURGLASS TIMER
30  HOME : HGR
40  HCOLOR= 3: GOSUB 310
50  GOSUB 150
60  HOME : VTAB 23: PRINT "PRESS ANY KEY TO START.";: GET
     GO$
70  HOME
90  GOSUB 210
100  VTAB 23: PRINT "PRESS Q TO QUIT.";: GET GO$: TEXT
110  IF GO$ <  > "Q" THEN 30
120  END
130  REM  * FILL  WITH  SAND *
140  REM   ------------------
150  FOR R = 100 TO 200
160  HPLOT 150,75 TO R,0
170  NEXT
180  RETURN
190  REM  * SIFT SAND THROUGH *
200  REM   -------------------
210  HPLOT 150,75 TO 150,150
215  FOR A = 1 TO 73
220  HCOLOR= 3
230  HPLOT 199,150 TO 150,150 - A TO 101,150
240  HCOLOR= 0
250  HPLOT 197,1 TO 150,A TO 103,1
260  FOR Z = 1 TO 200 - 3 * A: NEXT Z
270  NEXT A
280  RETURN
290  REM  * DISPLAY HOUR GLASS *
300  REM   --------------------
310  HPLOT 100,0 TO 200,0 TO 100,150 TO 200,150 TO 100
     ,0
320  RETURN
```

through an hourglass. The hourglass is constructed using the same technique used for displaying polygons. The sand is sifted from the upper portion of the hourglass to the lower portion using a technique similar to that used to draw the lines in the Color Sweep program.

Now let's look at each line of Hourglass Timer.

Lines 30-50: After clearing the text screen, line 30 displays and clears page 1 of high-resolution graphics. Line 40 then sets the color to white (3) and calls the subroutine at 310 to draw the hourglass. Line 50 calls the subroutine starting at line 150 to fill the top part of the hourglass with sand. Line 50 simply causes that subroutine to be executed.

Lines 60-90: These lines instruct the user to press any key to start the timer. The instructions must be printed on the bottom four lines of the text screen in order to be seen on page 1 of high-resolution graphics. Once a key is pressed in line 60, the instructions are erased by line 70, and the timer is started by calling the subroutine located in lines 210 through 280.

Lines 100-120: Line 100 instructs the user to press the Q key to quit. If any other key is pressed, the program starts over.

Lines 150-180: Filling the top portion of the hourglass with sand is accomplished by drawing lines from the center (150,75) to each point across the top of the hourglass. The points used to create the top of the hourglass extend from (100,0) to (200,0).

Line 210: A vertical line extending from the center of the hourglass to the bottom is plotted to begin the sifting process. This line represents the falling grains of sand.

Lines 215-280: These lines form the remainder of the "sifting sands" subroutine. White sand is placed in the bottom of the hourglass in line 230. A line is drawn from near the lower left corner to a point along the vertical centerline, then to a point near the lower right corner of the hourglass. The row coordinate of the point along the vertical centerline, used in constructing the lines which fill the bottom part of the glass, decreases by 1 each time sand is sifted. This gives the illusion of sand piling up in the bottom of the hourglass.

A similar technique is used to remove sand from the top of the hourglass in line 250; sand is removed by drawing black lines in the top part of the hourglass. The index limit of the timing loop in line 260 grows smaller each time sand is removed from the top and placed in the bottom of the hourglass, causing the sand to sift more quickly as the top of the hourglass empties.

Lines 310-320: This subroutine displays the hourglass. It is a simple design created by drawing two end lines and the diagonals of a rectangle. The resulting figure looks like two isosceles triangles joined at their peaks.

Other geometric figures often needed for creating graphic displays are curved lines, circles, and ellipses. As stated before, a good math background is needed to understand many of the techniques used in the remainder of this chapter. Some math concepts used will be discussed briefly when they are introduced.

Graphing equations using *cartesian* or *rectangular coordinates* is a topic that is studied in any algebra course. Points that satisfy the equations are plotted on graph paper to form straight lines, parabolas, and hyperbolas. The graph paper represents a plane (flat surface) divided into four sections, called *quadrants*, by two perpendicular lines or *axes* labeled X and Y (see Fig. 3-2). The point where the axes intersect or cross is called the *origin*. Each point in the plane can be identified by an ordered pair of numbers called its *coordinate*. The coordinate of the origin is the ordered pair (0,0). The horizontal position of the point in the plane is given by the first number of the ordered pair, and the vertical position by the second. Points to the left of the origin have a negative horizontal position, and points to the right of the origin have a positive horizontal position. Vertical coordinates are positive if the point is above the X axis and negative if it is below.

Paths of points (or *lines*) on the plane are described by equations like Y=2X+3. Any point on the line can be determined by replacing the X in the equation with a horizontal coordinate and then evaluating the expression to determine the value of

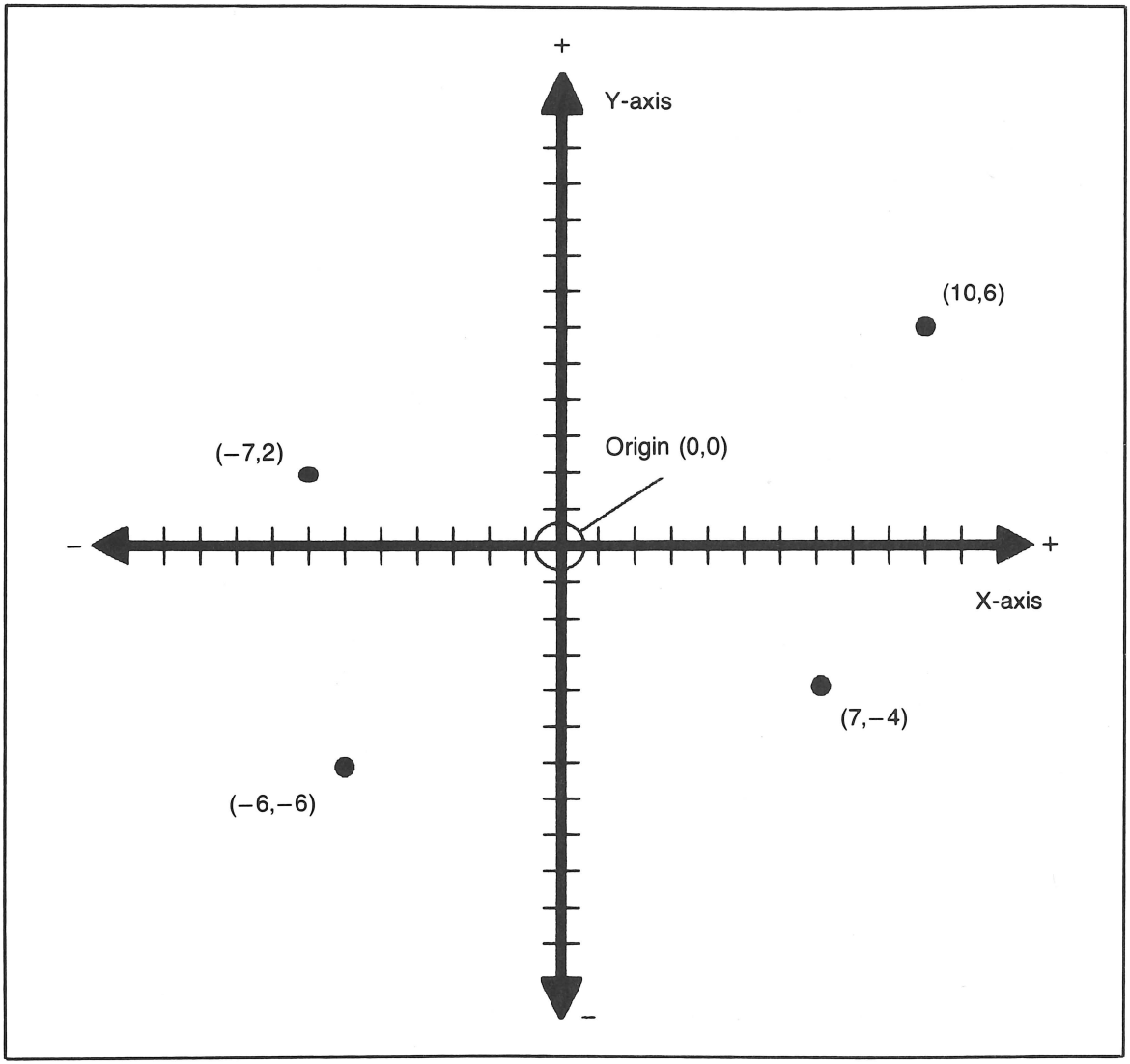


Fig. 3-2. The cartesian or rectangular coordinate system is used to identify the locations of points on a plane. In much the same way, column and row coordinates are used to identify the positions of points on the high-resolution graphics screen.

Y, the vertical coordinate of the point. For example, replace X in the preceding equation with the number 3. The value of Y is then 2 times 3 plus 3, which is 9. This means that (3,9) is the coordinate of a point on the line Y=2X+3. Once a point's coordinate is determined, it can be plotted on the plane. By plotting enough points, the line described by the equation is formed.

In order for the computer to graph equations, the rectangular coordinates obtained by evaluating the equation must be adjusted. The computer screen cannot display points with negative horizontal or vertical coordinates. Therefore the origin must be relocated, normally at the center of the screen; the coordinate of the origin consequently becomes (140,80). The coordinates obtained by evaluating the equation using the replacement method described above must be adjusted to allow

Listing 3-9. Equation Grapher.

```
10  REM   EQUATION GRAPHER
20  HGR : HCOLOR= 3
30 H = 140:K = 80
40  HPLOT 0,K TO 279,K: REM  HORIZONTAL AXIS
50  HPLOT H,0 TO H,159: REM  VERTICAL AXIS
60  FOR X =  - 20 TO 20: REM  CHOOSE X COORDINATE OF A
     POINT
70 Y = 2 * X + 3: REM  EVALUATE FOR CORRESPONDING Y CO
    ORDINATE
80  HPLOT H + X,K - Y: REM  PLOT THE POINT
90  NEXT X
100  END
```

for the new origin. After an ordered pair is determined, the X coordinate of the point must increase by the X coordinate of the origin (X + 140), and the Y coordinate of the point must be similarly adjusted. The Y coordinate of the origin, however, must *decrease* by the computed Y coordinate, because the vertical positions on the screen are numbered in the opposite direction from those used in the mathematical coordinate system. A program that will plot a segment of the line described by the equation Y=2X+3 is listed below. The horizontal coordinate of the origin is assigned to the variable H, and the vertical coordinate of the origin is assigned to the variable K.

Care must be taken when graphing lines with the computer so that both the X and Y coordinates of the point can be displayed on the screen. If the value of 140+X is greater than 279 or less than 0, or the value of 80–Y is greater than 191 or less than 0, an ILLEGAL QUANTITY error occurs. Inserting the following line allows the program to work for all values of X and Y:

```
75 IF ABS(X)>=H OR ABS(Y)>K THEN 90
```

The absolute value function, ABS, disregards the sign of the argument; therefore ABS(–5) is 5, and ABS(+5) is also 5.

Many equations can be graphed using the Equation Grapher program by changing line 70. Different segments of a line can be graphed by changing the loop parameters in line 60. If equations are selected that have no X and Y values that can be plotted on the screen, no graph is displayed, due to the check performed in line 75. Table 3-2 gives some equations and loop parameters that can be placed in line 70 of the Equation Grapher program, along with the type of graph that each should produce.

The sine and cosine functions are used frequently throughout the remainder of this chapter. These trigonometric functions are said to be *cyclic* because their Y values are repeated over and over. The Y values of both functions range from -1 through $+1$ during each cycle. The X distance required for a cyclic function to start repeating its Y values is the function's *period.* Figure 3-3 shows the graph of the sine and cosine functions over two periods from $X = -2\pi$ through $X = +2\pi$. Pi (π) is an *irrational number,* a decimal number whose exact value cannot be expressed. Pi is approximately 3.14, making 2π approximately 6.28. The values can, naturally, be expressed more accurately; the values given are accurate enough for the purpose of this book.

The Equation Grapher program can be used to graph the sine and cosine functions. Change lines 60 and 70 in the program to

```
60 FOR X = –6.28 TO 6.28
70 Y = SIN (X)
```

Table 3-2. Equation Forms for Equation Grapher Program.

Equation	Loop Parameters From	To	Graph
Y = 15	-60	+60	
Y = X	-70	+70	
Y = .1 * X ^ 2	-30	+30	
Y = 100 / X	+1	+100	
Y = ABS (X)	-80	+80	

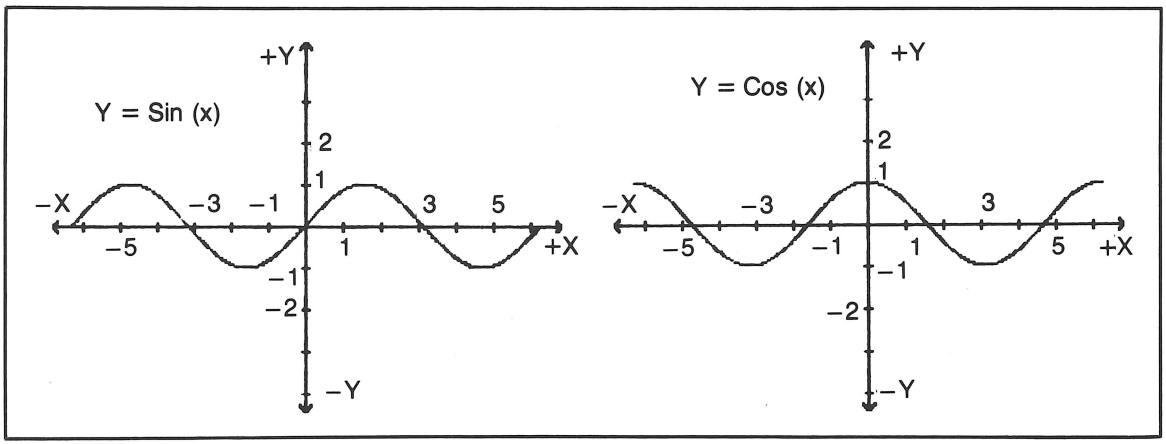


Fig. 3-3. The graphs of these functions repeat every 2π units (approximately 6.28).

to see a graph of the sine function. The cosine function can be graphed by changing line 70 to Y=COS(X) and using the line 60 given for the sine function.

Neither graph is very good when the stated modifications are made because not enough X values are used and the computed Y values are too close together. A better graph can be obtained by decreasing the STEP parameter in line 60 and introducing a scaling factor in lines 75 and 80. Replace the old lines 60, 75 and 80 with the ones listed below and add the given line 35 to improve the graph of the functions.

```
35 S = 10
60 FOR X=-6.28 TO 6.28 STEP .1
75 IF ABS(X)*S>=H OR ABS(Y)*S>=K
   THEN 90
80 HPLOT H+X*S, K-Y*S
```

Experiment with different values for the variable S until the desired scale is obtained. The STEP parameter in line 60 can be increased for plotting fewer points or decreased for plotting more points on the graph.

Familiarity with the sine and cosine functions is helpful in understanding programming applications using polar coordinates. *Polar coordinates* are also ordered pairs of numbers that described the positions of points on a plane. Instead of indicating the horizontal and vertical distances from the origin to a point, however, they give the straight-line distance from the origin to a point and the angle formed by such a line with respect to the positive X axis (see Fig. 3-4). The angle is measured in *radians* rather than degrees because the needed computer functions operate only in radian measures. A conversion chart for radians and degrees is given in Table 3-3.

Polar coordinates simplify the process of displaying circles on the high-resolution graphics screen. The distance from the origin to every point on the circle is the same—namely the radius of the circle. The angles describing the points on the circle do not need to be computed, since a point occurs at every angle T from 0 to 2π radians. Once the polar coordinates for a point are determined, they can be converted back to the more familiar X,Y coordinates using the following formulas (parametric equations):

X = Radius * cosine T
Y = Radius * sine T

A program that displays a circle with a radius of 30

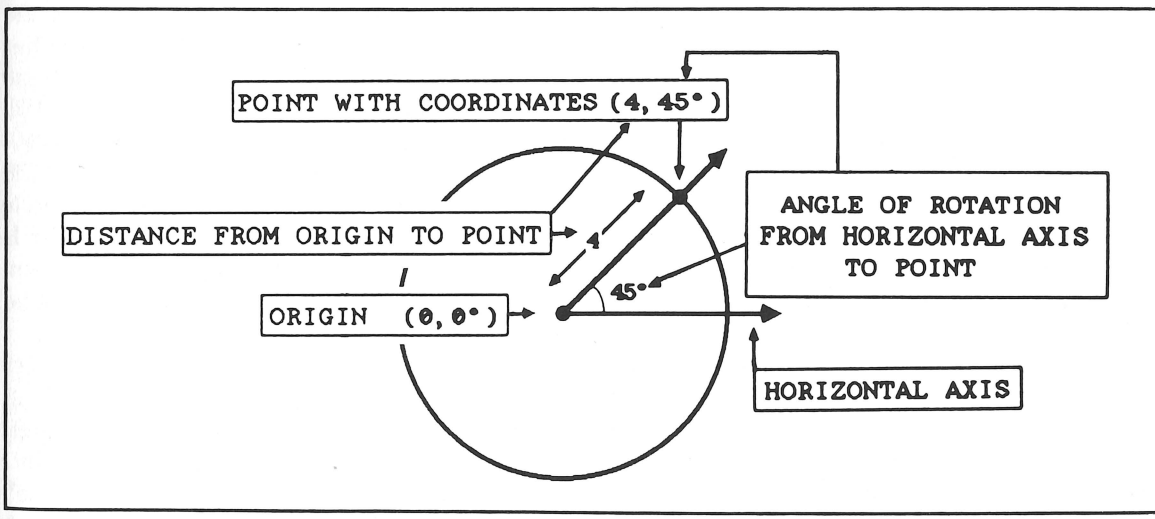


Fig. 3-4. The polar coordinate system identifies the locations of points on a plane by their distance from the origin and their angle of rotation from the horizontal axis. Polar equations are used to simplify programs that display complex patterns.

Table 3-3. Radian/Degree Conversion Chart.

Radians	Degrees
0	0
$\pi/6$	30
$\pi/4$	45
$\pi/3$	60
$\pi/2$	90
$2\pi/3$	120
$3\pi/4$	135
$5\pi/6$	150
π	180
$7\pi/6$	210
$5\pi/4$	225
$4\pi/3$	240
$3\pi/2$	270
$5\pi/3$	300
$7\pi/4$	315
$11\pi/6$	330
2π	360

Listing 3-10. Circle plotting program.

```
5  REM      CIRCLE
10  HGR
20  HCOLOR= 3
30 R = 30
40 H = 140
50 K = 80
60 TP = 6.28
70 C = R * TP
80  FOR T = 0 TO TP STEP TP/C
90 X =  COS (T) * R
100 Y =  SIN (T) * R
110  HPLOT H + X,K - Y
120  NEXT T
130  END
```

screen points appears in Listing 3-10.

Lines 30-50: The radius of the circle is assigned to the variable R. Increasing R increases the size of the circle. H is the variable used to store the horizontal coordinate of the center of the circle, and K is used for the vertical coordinate of the center.

Line 60: The value of 2π is assigned to the variable TP (Two Pi). It is a good idea to assign constants used repeatedly throughout a program to variables, because the computer takes more time to generate a number than it does to look it up in the variable table.

Line 70: The circumference of the circle is computed and assigned to the variable C. The formula for computing the circumference of a circle is

$$C=2\pi r$$

where r is the radius of the circle. TP is 2π so for the purpose of this program only TP*R need be calculated. The circumference is needed to indicate the number of screen dots required to form the desired circle.

Line 80: This line determines the angles at which the points on the circle must be plotted with respect to the X axis. A rotation of 360 degrees or 2π radians is needed to complete a circle; therefore the angle variable (T) must take on values from 0 to 6.28 (TP). The distinct number of angles needed for placing points on the circle is C, which is the number of points on the circle. TP/C is the step interval for the loop to plot C points on the circle.

Lines 90-100: Determining the horizontal coordinate of the point that is located R units from the center at the angle T from the X axis is accomplished in line 90. The horizontal coordinate is calculated and assigned to the variable X. Line 100 calculates the vertical coordinate for the point having the polar coordinates (R,T) and assigns it to the variable Y.

Line 110: This line plots a single point on the circle. The point is plotted at (H+X,K–Y) rather than (X,Y) because of the differences between the high-resolution screen coordinates and the actual plane coordinates.

Even if it is not clear how the points on the screen are computed in the program, the circle program can be useful. Changing the values for H and K causes the circle to be placed at different screen locations. Altering the value of R causes the size of the circle to be increased or decreased.

Ellipses can be displayed using a program very similar to the Circle program. Unlike circles, whose points are all equidistant from the center, the distance from the center of an ellipse to points on the periphery varies. In order to graph an ellipse, the radius R used for determining the X coor-

dinate and that used for determining the Y coordinate must have different values. Rather than use two radii, a horizontal scaling factor (HS) and a vertical scaling factor (VS) are introduced. The Ellipse program (Listing 3-11) displays an ellipse with a vertical scaling factor of 3 and a horizontal scaling factor of 4. Since the radius is 10, the ellipse has a horizontal span of 80 points and a vertical span of 60 points.

The Ellipse program differs from the Circle program in lines 20, 70, 90, and 100. Line 20 of the ellipse program assigns the horizontal and vertical scaling factor their respective variables. The HCOLOR=3 statement has been moved to line 10. The greater of the scaling factors is multiplied by the circumference of the circle of radius R in line 70 so that the figure will be closed. The conversion formulas for the X and Y coordinates are altered in lines 90 and 100 to include their respective scaling factors.

The Ellipse program can also be used to draw circles by assigning the same value, say 1, to both HS and VS. Since the same program lines can draw both circles and ellipses when appropriate scaling factors are chosen, a subroutine can be formed that can be used in any program that requires construction of ellipses, circles, or both. The circles and ellipses that have been constructed have their cen-

Listing 3-11. Ellipse plotting program.

```
5  REM      ELLIPSE
10  HGR : HCOLOR= 3
20 HS = 4:VS = 3
30 R = 10
40 H = 140
50 K = 80
60 TP = 6.28
70 C = R * TP * HS: IF VS > HS THEN C = R * TP * VS
80  FOR T = 0 TO TP STEP TP / C
90 X =  COS (T) * R * HS
100 Y =  SIN (T) * R * VS
110  HPLOT H + X,K - Y
120  NEXT T
130  END
```

Listing 3-12. Circle/ellipse subroutine.

```
999  REM   CIRCLE/ELLIPSE SUBROUTINE
1000 C = R * TP * HS: IF VS > HS THEN C = R * TP * VS
1010  HCOLOR= CO
1020  FOR T = S TO S + PI / 2 STEP TP / C
1030 X = R *  COS (T) * HS
1040 Y = R *  SIN (T) * VS
1050  HPLOT H - X,K - Y: HPLOT H + X,K + Y
1060  HPLOT H - X,K + Y: HPLOT H + X,K - Y
1070  NEXT T
1080  RETURN
```

ters at the origin; their graphs are said to be *symmetric about the origin*. Thus the number of calculations performed to find points on the figure can be reduced four times. When (H+X,K+Y) is a point on the figure, the following points also fall on the figure: (H−X,K−Y) (H−X,K+Y) and (H+X,K−Y). Since four points can be calculated for each value of the loop variable T, it need only take on values 0 through $\pi/2$. The subroutine (Listing 3-12) can be used in any program to construct circles and/or ellipses.

Before a program calls the subroutine the variables CO, R, TP, PI, S, VS, HS, H, and K must be assigned appropriate values for the figure desired. The variable CO stores the value for the color to be used for displaying the figure. The variable S is used to store the first angle for which a point on the figure is to be computed. All the sample programs up to this point have started plotting with the point obtained when 0 is the measure of the angle, but any number can be used. If S is nonzero, the final display is no different than if 0 is used. The figure is merely plotted starting at a different point on the figure. Add the Circle/Ellipse subroutine to the program segment in Listing 3-13 and run it. Experiment with different values for S to observe the effect of the changes.

Solid ellipses and circles can be displayed by changing two statements in the Circle/Ellipse subroutine. Replace lines 1050 and 1060 of the subroutine with the lines listed below.

```
1050 HPLOT H-X,K-Y TO H+X,K+Y
1060 HPLOT H-X,K+Y TO H+X,K-Y
```

When these lines are placed in the subroutine, a line segment connects points located across the figure from each other. All of these line segments pass through the center of the figure.

Listing 3-13. Program demonstrating use of the circle/ellipse subroutine.

```
5  REM   ONE EYE
10  HGR
20 S = 0
30 R = 10
40 TP = 6.28:PI = 3.14
50 K = 80:H = 140
60 CO = 3
70 VS = 3:HS = 5
80  GOSUB 1000
90 VS = 2.5:HS = VS
100  GOSUB 1000
110 VS = .5:HS = VS
120  GOSUB 1000
130  END
```

HIGH-RESOLUTION GRAPHICS APPLICATIONS

A program using many of the techniques discussed in this chapter is called Winter Word Game (Listing 3-14). In addition to demonstrating how all these ideas can be utilized in a program, Winter Word Game provides some relief from the mathematical discussions that have dominated the chapter. The game is a word guessing game similar to Hangman.

As discussed previously, the first step is to plan the rules for the game. For this game, the player is to guess the letters of a word; each letter guessed that does not appear in the word results in a penalty. In this case a portion of a snowman melts. If seven letters are selected that do not appear in the word, the snowman disappears and the player loses. If all the letters of the word are guessed before the seventh miss, the player wins. We suggest that the program be entered and executed before you study the program discussion.

Since the Winter Word Game program is rather lengthy and uses several variables, the discussion of the program is divided into two parts. The first part lists all the variables and arrays and indicates their respective roles. The second part explains the functions performed by a line or group of lines in the program. Many readers may be able to determine how the program works by merely referring to the variable list, while those with less programming experience may need to study both parts in detail.

The variables are listed in their order of appearance in the Winter Word Game program. Array variables are followed by a set of parentheses.

Listing 3-14. Winter Word Game program listing (continues through page 45).

```
10  REM   WINTER WORD GAME
20 NW = 15
30  DIM WD$(NW),USEDZ(26)
40  FOR X = 0 TO NW: READ WD$(X): NEXT
50  FOR X = 1 TO 26:USEDZ(X) = 0: NEXT
60  HOME : HGR
70  VTAB 22: PRINT "BUILDING A SNOW PERSON IS FUN.  WA
    TCH!"
80  HCOLOR= 6: HPLOT 0,0: CALL 62454
90 PI = 3.14159:TP = 2 * PI:S = PI / 2
100 HS = 1.2:VS = 1:H = 80:K = 120:R = 39:CO = 7
110  GOSUB 1000
120 HS = 1.5
130 K = 63:R = 25: GOSUB 1000
140 K = 25:R = 15: GOSUB 1000
150 HS = 1:K = 76:R = 6:H = 117:VS = 3:CO = 7: GOSUB 1
    000
160 H = 43: GOSUB 1000
170 CO = 4:R = 2:K = 20:H = 73:VS = 1:HS = 1.7: GOSUB
    1000
180 H = 87: GOSUB 1000
190 H = 82:HS = 1.2
200 K = 27:R = 2:CO = 5: GOSUB 1000
210  HCOLOR= 4:K = 33: HPLOT H - 2,K TO H + 10,K + 2
220  FOR K = 31 TO 35: HPLOT H + 7,K TO H + 10,K: NEXT
    K
230 HS = 1:CO = 4
240  FOR K = 45 TO 85 STEP 10
250  GOSUB 1000: NEXT K
260 MELT = 1: GOSUB 2100
270  PRINT : PRINT "BUT IT WON'T LAST LONG IN THE SUN.
    ";
280  HCOLOR= 4: FOR X = 30 TO 150 STEP 20: HPLOT 0,X TO
    10,X: NEXT
290 R = 25:H = 240:K = 25:VS = 1:CO = 7: GOSUB 1000
300 CR = 40:CO = 5: GOSUB 1010
310  FOR TIMER = 1 TO 2000: NEXT TIMER
320 MELT = 0
330 LG = 0
340  HOME : VTAB 21: PRINT "PREVENT THE SUN FROM MELTI
    NG THE SNOW": PRINT "PERSON BY GUESSING THE HIDDE
    N WORD.": PRINT : PRINT "PRESS <B> TO BEGIN.";
350  GET G$: IF G$ <  > "B" THEN 350
360 W =  RND (1) * (NW + 1):WD$ = WD$(W):L =  LEN (WD$
    )
370  HOME : VTAB 21: FOR X = 1 TO L: HTAB 10 + X: PRINT
     CHR$ (95);: NEXT X
380  VTAB 23: HTAB 1: PRINT "GUESS A LETTER: " CHR$ (8
```

```
     );: GET G$: PRINT G$
390 G =  ASC (G$) - 64
400  IF G < 1 OR G > 26 THEN 380
410  IF USEDZ(G) THEN 380
420 USEDZ(G) = 1
430 FOUND = 0
440  FOR X = 1 TO L: IF G$ <  >  MID$ (WD$,X,1) THEN 4
     70
450 FOUND = 1: VTAB 21: HTAB 10 + X: PRINT G$;
460 LG = LG + 1
470  NEXT X
480  IF  NOT FOUND THEN  GOSUB 2000
490  IF LG < L THEN 380
500 CO = 6: GOSUB 1000: HOME : FLASH : VTAB 22: PRINT
     WD$;: NORMAL : PRINT " - YOU WIN!!!"
510 WIN = WIN + 1
520  PRINT : PRINT "PRESS <P> TO PLAY OR <Q> TO QUIT";
530  GET G$: IF G$ = "P" THEN 50
540  IF G$ <  > "Q" THEN 530
550  HGR : TEXT : HOME
560  PRINT "YOU PLAYED "WIN + LOSS" GAME";: IF WIN + L
     OSS > 1 THEN  PRINT "S";
570  PRINT ".": PRINT : PRINT "YOUR RECORD IS: WINS =
     "WIN"  LOSSES = "LOSS"."
580  PRINT : PRINT : PRINT "YOU WON " INT (WIN / (WIN +
     LOSS) * 110 + .5)"% OF THE TIME."
590  END
1000 CR = R * TP * HS: IF VS > HS THEN CR = R * TP * V
     S
1010  HCOLOR= CO
1020  FOR T = S TO S + PI / 2 STEP TP / CR
1030 X = R *  COS (T) * HS
1040 Y = R *  SIN (T) * VS
1050  HPLOT H - X,K - Y TO H + X,K + Y
1060  HPLOT H - X,K + Y TO H + X,K - Y
1070  NEXT T
1080  RETURN
2000 MELT = MELT + 1
2010  GOSUB 2100
2020  IF MELT / 20 =  INT (MELT / 20) THEN  RETURN
2030  IF MELT < 148 THEN 2000
2040  POP
2050  HOME : VTAB 22: PRINT "THE WORD WAS "WD$"."
2060  FOR TIMER = 1 TO 2000: NEXT TIMER
2070  HCOLOR= 5: HPLOT 0,0: CALL 62454
2080 LOSS = LOSS + 1
2090  GOTO 520
2100  HCOLOR= 6: HPLOT 0,MELT - 1 TO 130,MELT - 1
2110  FOR B = MELT TO MELT + 7: HCOLOR= 6: HPLOT 0,B TO
     64,B: HPLOT 94,B TO 130,B: HCOLOR= 4: HPLOT 65,B TO
     93,B: NEXT
```

```
2120  FOR B = MELT + 8 TO MELT + 10: HCOLOR= 6: HPLOT
      0,B TO 58,B: HPLOT 100,B TO 130,B: HCOLOR= 4: HPLOT
      59,B TO 99,B: NEXT
2130  RETURN
2140  DATA  SNOW,SLEET,COLD,FREEZE,BLIZZARD,WINTER,ICE
      ,SKI,AVALANCHE,FROST,SLED
2150  DATA  ICICLE,DRIFT,ICY,SKATE,CHINOOK
```

NW	Number of Words stored in DATA statements.
WD$( )	WorDs array.
USED%( )	An integer array used for keeping track of which letters of the alphabet have been USED in guessing a word.
X	Loop variable.
PI	Variable stores the value of the constant π.
TP	Stores the value of the constant 2π.
S	The angle for computing the initial point for plotting the figure.
HS	Horizontal Scaling factor for ellipses.
VS	Vertical Scaling factor for ellipses.
H	Horizontal coordinate of the center of a circle or ellipse.
K	Vertical coordinate of the center of a circle or ellipse.
R	Radius of the circle or ellipse.
CO	COlor used for displaying a figure.
MELT	The vertical coordinate of the portion of the snowman that is to be MELTed.
CR	CiRcumference of the circle, modified by the elliptical scaling factor with radius R.
TIMER	Variable for TIME delay loop.
LG	Number of Letters Guessed that are in the word.
G$	Stores keystrokes for controlling program flow, as well as the letter guessed by player.
W	Random value indicating the position of the word to be guessed in the WD$( ) array.
WD$	String variable which stores the WorD that is to be guessed.
L	Length of the word that is to be guessed.
G	A number from 1 to 26 representing the letter Guessed.
FOUND	A flag set if the letter guessed is FOUND in the word.
WIN	Counter which totals the number of games the player WINs.
LOSS	Counter which totals the number of games the player loses.
T	The angle for calculating the coordinates of a point on the circle or ellipse.
B	Loop variable indicating the vertical coordinate of the line segments used to melt the snowman.

Techniques used in some portions of the Winter Word Game program have been discussed in detail earlier in the chapter. Explanations concerning those sections of the program are very brief.

Line 20: The number of words contained in the DATA statements, less 1, is assigned to the variable NW. There are 16 words in the DATA statements, so NW is assigned a value of 15. The number is used to indicate the greatest subscript needed for the WD$( ) array. Since the 0 element of the array is used, 15 is the greatest subscript needed.

Lines 30-40: This line dimensions the WD$( ) array to hold the 16 words stored in DATA statements. A word can be chosen randomly more easily from an array than it can from DATA statements. The integer array USED%( ) keeps track of which letters of the alphabet have been used in trying to guess the word. An element value of 0 indicates the letter has not yet been used. The

element is assigned a value of 1 after the letter has been guessed. Elements 1 through 26 are used to indicate the use of the letter A through Z respectively. Line 40 assigns the words stored in DATA statements to the WD$( ) array.

Line 50: The values of all elements in the USED%( ) array must be set to 0 before each game to indicate that none of the letters of the alphabet have been used for guessing the word.

Lines 60-80: This sequence of lines first clears the text screen and then displays and clears page 1 of high-resolution graphics. A message is placed on the 22nd line of the text screen for the player to read while the snowman is being displayed. Then the background color is set to blue.

Line 90: This line assigns the constants π (3.14159), 2π, and $\pi/2$ to the variables PI, TP, and S respectively. The value 3.14 can be used in place of the more precise number without affecting the graphics displays.

Lines 100-110: The variable assignments needed in the Circle/Ellipse subroutine (line 1000) for displaying the lower section of the snowman are made in line 100. The color code 7 is used for white rather than 3 since the background color is blue (6). Colors in the range 0 through 3 do not always blend well with colors in the range 4 through 7. The subroutine that causes the ellipse to be displayed is called in line 110.

Lines 120-160: These lines cause the midsection of the snowman to be displayed. Only the variables that need to be changed are assigned new values. The head of the snowman is displayed after line 140 is executed. Lines 150 and 160 display the right and left arms, respectively, of the snowman.

Lines 170-180: Black eyes are placed on the face of the snowman using the color 4—rather than the less compatible color 0. The left eye is displayed in line 170 and the right in line 180. (Notice that the eyes do not appear to be elliptical. When small radii are used for circles or ellipses, the resulting figures appears to be rectangular.)

Lines 190-250: An orange nose is placed on the snowman's face. Line 210 draws a black line which serves as the mouth of the snowman and a stem for a pipe that is placed in its mouth. Line 220 contains a loop that displays a rectangle which represents the bowl of the snowman's pipe, and lines 230-250 draw small black circles (which appear to be squares) down the front of the snowman, positioned at vertical increments of 10 screen points.

Line 260: The variable MELT is used to indicate the vertical coordinate of the portion of the snowman that is melted away by an incorrect guess. Melting is accomplished by lowering the hat. The subroutine at line 2100 places the top of the hat at the vertical position stored by the variable MELT. Line 260 simply instructs the computer to display the hat starting at the vertical coordinate 1. Since no part of the snowman is pictured this high on the screen, the line simply places the hat on top of the snowman's head before the start of the game.

Line 270: This line displays a message announcing the impending sunrise at the bottom of the screen. Since the sentence is printed on line 24 of the text screen, it must be followed by a semicolon to prevent the text portion of the screen from scrolling.

Line 280: The loop in line 280 displays a series of short horizontal lines on the left edge of the screen at intervals of 20 dots, starting at vertical position 30. These marks indicate the number of incorrect guesses the player has remaining before losing the game. The lines disappear as the snowman melts.

Lines 290-300: The white circle displayed by line 290 clears a portion of the screen for displaying the sun. A solid red circle was not desired for representing the sun, so only 40 points on the circle were used to give a radiation effect to the picture. This was accomplished in line 300 by setting the circumference (CR) to 40 and calling the portion of the Circle/Ellipse subroutine that begins at line 1010. If GOSUB 1000 had been used in this statement, the desired value for CR would have been replaced by the value computed in line 1000.

Lines 310-330: Line 310 contains a time delay loop so the player can finish reading the messages at the bottom of the screen. Line 320 reinitializes MELT before the start of the game, so that the melting process begins with the proper

vertical coordinate. Before starting the game, the variable that represents the number of correct guesses must be set to 0. Line 330 makes the reset.

Lines 340-350: Line 340 clears the text portion of the screen and prompts the player to press the letter B to start the game. The player's response to the prompt is obtained by the statement GET G$ in line 350. If a key other than B is pressed, it is ignored. The game does not begin until the letter B is pressed.

Line 360: A number from 0 through 16 is randomly selected and assigned to the variable W. The word stored in the Wth element of the WD$( ) array is assigned to the string variable WD$. The word selected is the word the player must try to guess. The number of letters in the word is determined by the LEN function and assigned to the variable L.

Line 370: An underline character is placed on the screen for each letter in the word that is to be guessed. Since the underline character is not available from the keyboard, it must be referenced by its ASCII code, 95. The underline characters are placed on the 21st text screen line, starting in the 11th horizontal position.

Line 380: The player is prompted to take a guess in line 380. A space is printed immediately after the colon at the end of the sentence to erase any previous letters that may have been guessed. CHR$(8) is the backspace character. It is printed following the prompt statement so that the next letter guessed can be placed in the location cleared by the space at the end of the prompt statement. When a guess is taken, it is assigned to the variable G$ and printed on the screen.

Line 390: Sixty-four is subtracted from the ASCII code of the letter guessed and assigned to the variable G. The subtraction is performed because the ASCII codes for the letters of the alphabet are in the range 65 to 90, but the program needs to have them in the range 1 to 26 in order to keep track of which letters have been used.

Line 400: If the key pressed is not a letter, then its G value calculated in line 390 will not fall in the range 1 to 26. This line prevents the player from being penalized for pressing a key that does not represent a letter of the alphabet.

Line 410: A check is performed in line 410 to see if the letter chosen has already been guessed. If this check were not done, then determining if the game has been won or lost would be impossible. If the letter has already been guessed, the player must make another selection. If the element of the USED%( ) array that corresponds to the letter guessed is 0, then the letter has not yet been guessed.

Lines 420-430: Once it has been determined that the letter has not been used for a guess, the element which corresponds to the letter guessed is flagged (set to 1) in line 420. FOUND is the variable that acts as a flag to indicate whether the letter guessed is in the word. The flag must be cleared (set to 0) in line 430 before the letter search begins.

Lines 440-470: These lines search the word to see if it contains the guessed letter. The letter guessed is compared to each letter in the word by line 440, using the MID$ function. The arguments for the function are WD$, the word or string being checked; X, the position within the word that is to be checked; and 1, the number of letters in the string that are to be compared. If the guessed letter is not found in the position evaluated, a branch is taken so the next position can be checked. If the letter guessed is in the word, the FOUND flag is set. The letter is then printed on the screen, replacing the underline character that was used to mark its place in the word. Line 460 increments the counter LG, which keeps track of the number of correct guesses.

Line 480: The logic of this line requires some explanation. If the guessed letter was not found in the word the FOUND flag is still clear. The IF-THEN statement evaluates the condition NOT FOUND and determines its truth value. If the condition is true its truth value is 1; if false, its truth value is 0. If the truth value is 1 the THEN portion of the IF-THEN statement is executed, but if the condition's truth value is 0 the instructions following THEN are ignored. NOT is a *logical operator* that reverses the truth value of an expression. Therefore, if the FOUND flag is clear (false), then NOT FOUND is true. This statement causes a

branch to the melting routine at line 2000 if the letter did not appear in the word.

Line 490: This line determines if all the letters of the word have been guessed. If they have not, a branch is taken to allow another guess to be taken.

Lines 500-510: These lines are executed only when the player wins the game by guessing all the letters of the word. The circular area of the screen where the sun is displayed is changed to the background color by displaying a blue circle over the orange and white area. The variable WIN counts the number of games won.

Lines 520-590: Players can play another game by pressing the letter P, or exit by pressing. the letter Q. If a player elects not to play any more games, lines 550-590 give the scores and end the game. The percentage of games won is rounded to the nearest whole number.

Lines 1000-1080: This is the Circle/Ellipse subroutine presented earlier in the chapter. Lines 1050 and 1060 have been modified for displaying filled circles.

Lines 2000-2030: Melting the snowman is controlled by this subroutine. A single row of the snowman is removed at a time and replaced with the hat. Twenty rows of the snowman disappear each time the subroutine is used. MELT stores the vertical coordinate of the row that is to be removed. If MELT is divisible by 20, the melting stops and the game continues. If MELT has a value of 149, then the snowman has been melted away and the game is over. The subroutine at line 2100 accomplishes the melting.

Lines 2040-2090: When the snowman disappears, the player loses. These lines announce the unhappy event and then change the background color of the screen to orange. The POP statement in line 2040 cancels the GOSUB that called the melting subroutine the last time. If POP is omitted and the player continues to play the game, an OUT OF MEMORY error occurs. Line 2080 counts the player's losses. After the loss, a branch is taken to offer the player another game.

Lines 2100-2130: This subroutine displays the snowman's hat. In addition, it removes anything that resides between columns 0 and 130 on the screen by extending a blue horizontal line from either side of the hat. Line 2100 places a blue line over the top line of the hat, line 2110 redraws to top of the hat, and line 2120 redisplays the brim of the hat. MELT is the vertical coordinate of the second row of the hat, and MELT+10 is the vertical coordinate of the bottom of the hat.

Lines 2140-2150: The words used to play the game may be changed by placing new words in the DATA statements. The program functions properly only if the words entered have no spaces or hyphens, but could be modified to accept any word.

Listing 3-15. Five Petal Rose.

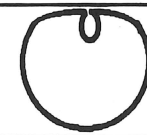
```
5  REM   FIVE PETAL ROSE
10  HGR : HCOLOR= 3
20 HS = 10:VS = 10
30 H = 140
40 K = 80
50 TP = 6.28
60 C = .1
70  FOR T = 0 TO TP STEP C
80 R = 2 *  COS (5 * T)
90 X =  COS (T) * R * HS
100 Y =  SIN (T) * R * VS
110  IF  ABS (X) > H OR  ABS (Y) > K THEN 130
120  HPLOT H + X,K - Y
130  NEXT T
```

Polar equations can be extremely useful for creating high-resolution graphics displays of designs other than circles and ellipses. The program in Listing 3-15 displays a figure called a five-petal rose; it is very similar to the Ellipse program presented earlier in this chapter. The primary difference is that the value of the radius (R) is not a constant in the Five-Petal Rose program. R is computed for each angle (T) for which a point is displayed. Therefore the assignment statement for R must be placed in the display loop. Another difference is that a constant value has been assigned to the circumference variable (C). Computing the number of points needed to plot a solid curve from polar coordinates involves the use of calculus. Most readers will find that it is easier to obtain the desired output by experimenting with the value of C in line 60, rather than by trying to compute the arc length of the graph.

The Five-Petal Rose program can be used to display the graphs of nearly any polar equation by placing the desired equation in line 80. A list of polar equations and the curves obtained by their graphs is presented in Table 3-4. Textbooks on analytical geometry are another source of polar equations that may be used in the program.

Although a method for generating displays of polygons by using the HPLOT-TO statement was discussed previously, a more sophisticated approach for displaying regular polygons can now be introduced. *Regular polygons* are polygons whose sides are all the same length. The technique makes use of the fact that any regular polygon can be inscribed in a circle. This means that a circle can be drawn around the polygon in such a way that each vertex or corner of the polygon lies on the circumference of the circle. A polygon of S sides can be constructed by connecting S equally spaced points

Table 3-4. Polar Equation Forms for Five-Petal Rose Program.

POLAR EQUATION	GRAPH
R = .5 * T	
R = 1 + COS (2*T)	
R = 1 - .5 * SIN (3*T)	
R = .5 - 1 * COS (T)	
R = .5 * T * COS (T)	

on the circle with straight lines. The program Polygons, (Listing 3-16) displays regular polygons having 3 to 15 sides. After each polygon is displayed, a keypress is required to view the next polygon.

Listing 3-16. Polygons.

```
10  REM     POLYGONS
20 TP = 6.28
30 H = 140:K = 80
40 R = 30
50  HCOLOR= 3
60  FOR S = 3 TO 15
70  HOME : HGR
80  VTAB 22: PRINT "NUMBER OF SIDES = "S".";
90  HPLOT H + R,K
100  FOR T = 0 TO TP STEP TP / S
110 X =  COS (T) * R
120 Y =  SIN (T) * R
130  HPLOT  TO H + X,K - Y
140  NEXT T
150  HPLOT  TO H + R,K
160  GET G$
170  NEXT S
180  END
```

Variable usage in the Polygons program is consistent with other programs in this chapter that have displayed circles and ellipses. A new variable S is used to store the number of sides of the polygon being displayed.

Lines 20-50: R is the radius of the circle which would inscribe the polygons, H and K as the horizontal and vertical coordinates of the center of that circle, and TP as 2π. The polygons are displayed using the color white (3).

Lines 60-80: S is the loop variable that takes on the values for the number sides of the polygons. Regular polygons with 3 to 15 sides will be displayed. Line 70 clears the text screen and also clears and displays page 1 of high-resolution graphics. Line 80 displays the number of sides for the polygon at the bottom of the screen.

Lines 90-120: The first vertex of the polygon is plotted. If this statement is omitted, no graphics are displayed due to the form of the HPLOT-TO statement in lines 130 and 150. Coordinates of only S points need to be calculated to construct the polygon, so the angles used are $2\pi/S$ radians apart. The necessary rectangular coordinates are computed in lines 110-120.

Lines 130-140: This sequence displays a line segment joining the current point and the most recently plotted point. The process continues until angle T has taken on all necessary values.

Line 150: Closing the polygon is accomplished by connecting the last point plotted to the first point plotted. Many of the polygons can be displayed without this statement, but some will have a missing side. This is due to the fact that π is an irrational number that cannot be expressed precisely.

Lines 160-170: Each polygon remains on the screen until a key is pressed, indicating that the next polygon is to be displayed.

Displaying solid polygons is not as easy as displaying solid circles or ellipses. A line segment joining the center with each point on the perimeter of the polygon must be constructed. The difficulty in performing this task is that the coordinates for all

the points that lie on perimeter of the polygon are not available when the preceding technique is used. HPLOT-TO does not provide any feedback on the coordinates of the points used to connect the endpoints of the segment. In order to shade the polygon an *algorithm* or method must be developed that computes the coordinates for each point used in the construction of the polygon.

One way of determining the coordinates of the points that lie between any two adjoining vertices on the polygon is to determine the equation of the line formed by connecting them. Two values, *slope* and *y-intercept*, are needed to arrive at the equation for a line. Slope is the ratio of the vertical change to the horizontal change between any two points on the line. Given two points with coordinates (x1,y1) and (x2,y2), the slope of the line on which they lie is (y2-z1) /(x2-x1). The y-intercept (B) can be determined once the slope (SL) is known by using the following formula:

$$B = y1 - x1 * slope$$

For example, if (100,120) are the coordinates for one endpoint and (130,60) are the coordinates of the other endpoint, then the slope is (120–60)/(100–130) which is 60 /–30 or –2. The y-intercept for the line is 120–100*(–2), which is 120+200 or 320.

The Solid Polygons program (Listing 3-17) utilizes the information in the previous paragraph to display solid polygons with 3 to 15 sides.

Lines 10-50: Polygons are displayed using the color white (3). The value 2π is assigned to TP. SP is the angle at which the first vertex will be placed; changing the value of SP changes the orientation of the polygon on the screen. H and K

Listing 3-17. Solid Polygons.

```
10  REM      SOLID POLYGONS
20  HCOLOR= 3
30 TP = 6.28:SP = 0
40 H = 140:K = 70:R = 50
50 HS = 2:VS = 1
60  FOR S = 3 TO 20
70  HOME : HGR
80  GOSUB 2000
90  VTAB 23: PRINT S" SIDES.  PRESS ANY KEY TO CONTINU
    E.";: GET G$
100  NEXT S
110  END
2000  FOR T = SP TO SP + TP - (TP - 1) / S STEP TP / S
2010 X1 =  COS (T) * R + H:Y1 =  -  SIN (T) * R + K
2020 X2 =  COS (T + TP / S) * R + H:Y2 =  -  SIN (T +
    TP / S) * R + K
2030 SL = (Y2 - Y1) / (X2 - X1)
2040 B = Y1 - SL * X1
2050 ST =  SGN (X2 - X1)
2060  IF  ABS (SL) > 1 THEN ST = ST / ( ABS (SL) + 1)
2070  FOR X = X1 TO X2 STEP ST
2080 Y = SL * X + B
2090  HPLOT X,Y TO H,K
2100  NEXT X,T
2110  RETURN
```

are the variables for the center of the polygons, and R is the variable for the radius of the circle that would inscribe them. HS and VS represent the horizontal and vertical scaling factors. When HS and VS are the same value, regular polygons are formed.

Lines 60-70: A loop for displaying solid polygons with three to fifteen sides is initialized in line 60. The variable S stores the number of sides. Then, in line 70, previous displays are erased from the screen.

Lines 80-110: The subroutine that constructs a solid polygon with S sides is called. After the polygon is displayed, its number of sides is printed at the bottom of the screen. A key press is required before the next polygon is displayed.

Line 2000: The subroutine that constructs a solid polygon starts at line 2000. The values of the angles needed to plot the vertices of the polygons are assigned to the variable T. The loop limit is reduced by the expression (TP-1)/S, since two consecutive points are calculated in the loop. If this adjustment is omitted, the first section of the polygon will be drawn twice, extending the time required to display the figure. The number of angles needed for constructing each polygon is S (sides), so the step parameter is calculated accordingly.

Lines 2010-2020: The coordinates (X1,Y1) for the initial endpoint of a side are calculated and assigned to the appropriate variables. Line 2020 computes coordinates for the terminal endpoint of a side and assigns them to the variables X2 and Y2. Two endpoints are needed so the equation of the line which forms a side of the polygon can be derived.

Lines 2030-2040: Before a line can be constructed between points (X1,Y1) and (X2,Y2) the equation of the line must be determined. The slope (SL) of the line is calculated in line 2030 and the y-intercept is calculated in line 2040.

Lines 2050-2060: The step parameter (ST) for the loop that displays the section of the polygon is calculated here. The sign of the step is determined first, then the size of the step is computed.

Lines 2070-2080: A loop is established that generates the horizontal coordinates (X) of all points that are on the current side of the polygon. The vertical coordinate (Y) for each point on the current side of the polygon is computed in line 2080 using the linear equation derived in lines 2030 and 2040.

Lines 2090-2110: A line segment connects the center (H,K) of the polygon to a point on the perimeter (X,Y). When all the points along the current side have been connected with the center, the remaining segments are constructed. The subroutine is then excited.

The Solid Polygons program can be used to construct outlines of polygons by changing line 2090 to HPLOT X,Y.

PAGE FLIPPING

Several algorithms for creating high-resolution displays have been presented in this chapter. Mathematics has been used extensively to reduce the number of program lines needed to form the desired figures. Unfortunately, the techniques used are often slow because of the speed at which the computer makes calculations. There is no easy way to rectify this situation when the programmer prefers that the user observe the step-by-step construction of the picture. If the order in which portions of the picture appear on the screen is not important, however, an alternate technique can be used.

The procedure requires that the graphics displays be created by a program separate from the one that delivers them. The program that creates the displays can utilize any or all of the programming techniques presented in forming the desired picture. The memory segment corresponding to the high-resolution graphics page used to display the picture is stored in a *binary file* on the disk. The program that delivers the graphics scene loads the information back into the appropriate memory segment from the disk, causing the picture to be displayed. Although the process may seem confusing, it is really quite easy.

The Polar Design Maker program (Listing 3-18) creates the artistic display of Polar Design I and Polar Design II shown in Fig. 3-5. Use of the designs would normally be extremely limited, since

Listing 3-18. Polar Design Maker.

```
10  REM  POLAR DESIGN MAKER
20  HGR2
30 YES = 1:NO = 0
40 CO = 3:HS = 1.4:VS = 1:H = 140:K = 90
50  FOR F = 4 TO 80 STEP 2
60  DEF  FN POLAR(T) = F -  COS (T)
70 SA = F / 10:LA = SA + 4.71
80 NP = F * 6.28 * HS
90  GOSUB 320
100  NEXT F
110  FOR F = 82 TO 90 STEP 2
120 SA = F / 10:LA = SA + 6.28
130  DEF  FN POLAR(T) = F
140 NP = F * 6.28 * HS
150  GOSUB 320
160  NEXT F
165  PRINT  CHR$ (4)"BSAVE POLAR DESIGN I,A$4000,L$200
     0"
170  DEF  FN POLAR(T) = 9 - 20 *  COS (T + .785)
180 SA = 0:LA = SA + 6.28:H = 30:K = 27:HS = 1
190 NP = 200
200  GOSUB 320
210  DEF  FN POLAR(T) = 20 *  SIN (3 * T - .785)
220 LA = SA + 3.15:FILL = YES
230 H = 258:K = 17: GOSUB 320
240 NP = 575:FILL = NO
250 K = 170: DEF  FN POLAR(T) = 20 *  COS (5 * T + .62
    8): GOSUB 320
260 FILL = YES
270 H = 28:K = 167: DEF  FN POLAR(T) = 8 - 20 *  SIN (
    2 * T)
280 LA = LA * 2
290 FILL = 1:NP = 220
300  GOSUB 320
310  PRINT  CHR$ (4)"BSAVE POLAR DESIGN II,A$4000,L$20
     00"
315  END
320  HCOLOR= CO
330  FOR T = SA TO LA STEP 6.28 / NP
340 R =  FN POLAR(T)
350 X = R * HS *  COS (T)
360 Y = R * VS *  SIN (T)
370  HPLOT H + X,K - Y: IF FILL THEN  HPLOT  TO H,K
380  NEXT T
390  RETURN
```

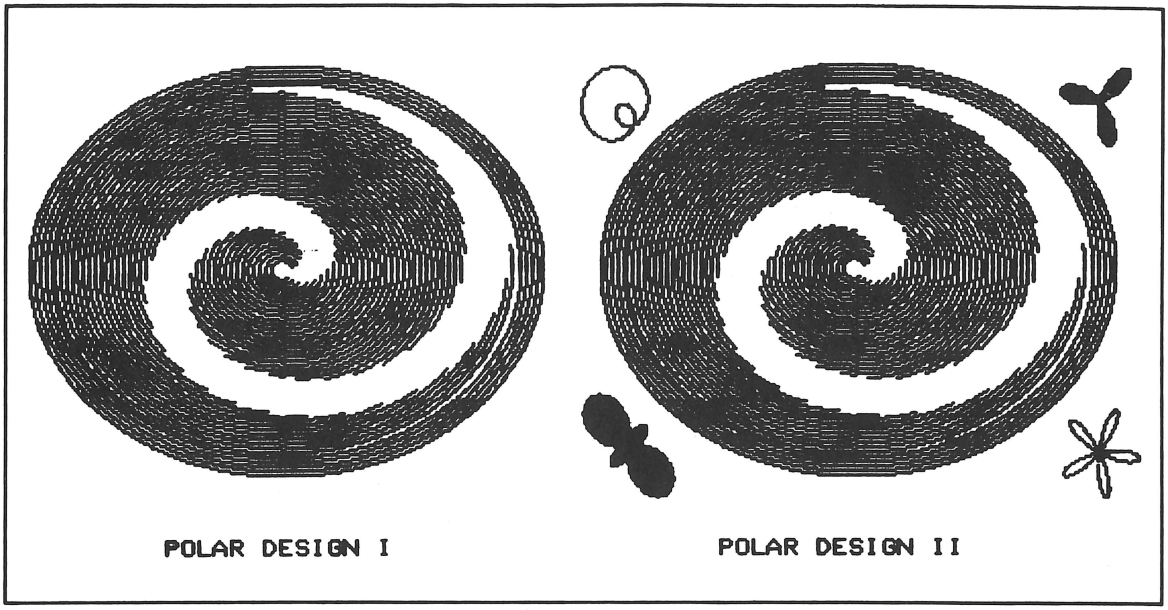


Fig. 3-5. These patterns are created by the Polar Design Maker program.

the execution time of the program is approximately 25 minutes. However, once the Polar Design Maker program is run, its resultant graphics designs can subsequently be displayed in just a few seconds. Before entering the program, boot the disk operating system (DOS) and place an initialized diskette in the disk drive. Save the program, then run it.

The program creates the inner swirl of the design and then stores the picture in a binary file called Polar Design I. After the swirl portion is saved in line 165, the polar ornaments surrounding the swirl are formed, and the entire picture is saved in another binary file called Polar Design II.

Any segment of the computer memory can be placed in a binary file by using the BSAVE command. The general form of the command is

BSAVE {filename}, {address}, {length in bytes}

Any filename can be used as long as it begins with a letter of the alphabet and does not contain commas or colons, but a comma *must* follow the filename. The memory address where the information to be saved is located is then specified in either decimal (base 10) or hexadecimal (base 16) numbers. The memory address parameter must be preceded by the letter A; if the address is expressed as a hexadecimal number, it must be preceded by a dollar sign ($). Another comma is used to separate the address and length parameters. The length parameter is preceded by the letter L. The number of bytes of memory that are to be placed on the disk is then specified in either decimal or hexadecimal form. Hexadecimal numbers indicating the length must also be preceded by a dollar sign.

Any DOS command such as BSAVE may be used in an Applesoft program by placing it in a PRINT statement. The PRINT statement must be of the form

PRINT CHR$(4)"{DOS command}"

The CHR$(4) signals the computer that the information enclosed in the quotation marks is a DOS command. The computer will then perform the instruction within the quotes rather than PRINT it on the screen.

The Polar Design Maker displays its output on high-resolution graphics page 2, which starts at

Listing 3-19. Display Design program.

```
10  REM   DISPLAY DESIGN
20  HGR2
30  PRINT  CHR$ (4)"BLOAD POLAR DESIGN II"
```

memory location 16384 (or $4000 hexadecimal). Each page uses 8192 bytes of memory, which is $2000 in hexadecimal form. Two files, Polar Design I and Polar Design II, are saved in lines 165 and 310 respectively.

Either picture may be placed back into memory through the use of the BLOAD command. The general form of the BLOAD command is

BLOAD {filename}, {address}

The address parameter specifies the memory location that is to receive the first byte of information from the binary file. If the information is to be placed back in the same memory segment from which it was BSAVEd, no address parameter is needed. Note that the BLOAD command has no length parameter. As is the case with any DOS command, it must be placed in the PRINT CHR$(4) statement if it is to be used within a program.

The program Display Design (Listing 3-19) will display Polar Design II on high-resolution graphics page 2. Page 2 is displayed and cleared in line 20. When line 30 is executed, the Polar Design II file is placed in the memory section from which it was saved. The picture fades in as the file is loaded.

Fade-in can be avoided by loading the picture before the graphics screen is displayed. Both HGR and HGR2 clear their respective graphics screens. Therefore, the graphics screens must be displayed using an alternate method if they are not to be erased. The screen displayed can be controlled by setting one or more switches that can be altered by a program. Such switches are called *soft switches* because they can be controlled by software instructions. References to certain memory locations are used to set the switches.

No-Fade Display (Listing 3-20) is a program which will display Polar Design I on high-resolution graphics page 2 without the fade-in visual effect.

The switch located at – 16304 sets the display mode to graphics. Location – 16302 sets the display mode for full-page graphics screen. Referencing location – 16299 causes a switch from page 1 to page 2. The final step is to trigger the switch at – 16297, which causes the computer to display the page indicated by the settings of the other switches.

Although a picture may have been placed in a binary file from page 2, it can be displayed on page 1. The Page 1 Display program (Listing 3-21) loads the Polar Design II file into the memory segment for page 1 graphics and displays the picture using the no-fade approach.

Since Polar Design II is placed in a different

Listing 3-20. No-Fade Display.

```
10  REM   NO FADE DISPLAY
20  PRINT  CHR$ (4)"BLOAD POLAR DESIGN I"
30  POKE  - 16304,0
40  POKE  - 16302,0
50  POKE  - 16299,0
60  POKE  - 16297,0
```

Listing 3-21. Page 1 Display program.

```
10  REM   PAGE 1 DISPLAY
20  PRINT  CHR$ (4)"BLOAD POLAR DESIGN II,A$2000"
30  POKE  - 16304,0
40  POKE  - 16302,0
50  POKE  - 16300,0
60  POKE  - 16297,0
```

memory segment than where it was saved, the address parameter of the new location must be specified. High-resolution graphics page 1 memory begins at memory location 8192 or $2000. POKE −16300,0 in line 50 tells the computer to display page 1.

Both pages of high-resolution graphics memory may contain pictures. Either picture can then be placed on the screen simply by setting the proper soft switches. Alternating between the pages can result in some interesting displays. The technique is called *page flipping* and is demonstrated in the program Alternating Designs (Listing 3-22). When the program is executed, Polar Design I is placed in the page 1 memory segment and Polar Design II is placed in the page 2 memory segment. The pictures are displayed alternately 100 times, with time loops controlling the tempo.

Page flipping is also useful when several pictures are to be displayed. A "slide show" can be presented by creating several binary files containing high-resolution graphics pictures. The program that delivers the slide show can load a picture into the page 1 memory segment and display it while loading another picture into the memory segment for page 2. When the user has viewed the picture on page 1, the pages can be flipped and another picture can be loaded into page 1 while the user views page 2. The process can be continued until all the pictures have been displayed.

A Slide Show program appears in Listing 3-23. In order to use the program, the disk containing the Polar Design I and Polar Design II picture files must be in the disk drive. The names of the picture files are stored in the DATA statement in line 240; any picture file may be displayed in the slide show by placing the desired filenames in DATA statements. If more than two pictures are to be displayed, the variable for the number of pictures (NP) in line 30 must be assigned the appropriate value. The pictures are shown using the entire screen for graphics display, so no text lines are available at the bottom of the screen.

Another advantage of storing graphics pictures

Listing 3-22. Alternating Designs program demonstrates page flipping.

```
10  REM   ALTERNATING DESIGNS
20  PRINT  CHR$ (4)"BLOAD POLAR DESIGN I,A$2000"
30  PRINT  CHR$ (4)"BLOAD POLAR DESIGN II"
40  POKE  - 16304,0
50  POKE  - 16302,0: POKE  - 16297,0
60  FOR FLIP = 1 TO 100
70  POKE  - 16300,0
80  FOR TIME = 1 TO 500: NEXT TIME
90  POKE  - 16299,0
100  FOR TIME = 1 TO 500: NEXT TIME
110  NEXT FLIP
120  TEXT : END
```

Listing 3-23. Slide Show.

```
10  REM   SLIDE SHOW
20  HGR : HGR2 : TEXT
30 NP = 2
40 ML =  - 16298
50 P = 2
60  HOME : HTAB 8: PRINT "SLIDE SHOW PRESENTATION"
70  READ F$
80  PRINT  CHR$ (4)"BLOAD"F$",A"8192
90  PRINT : PRINT : PRINT "PRESS A KEY TO START THE SL
    IDE SHOW."
100  PRINT : PRINT "CHANGE PICTURES BY PRESSING ANY KE
    Y."
110  PRINT : GET GO$: PRINT
120  POKE  - 16304,0: POKE  - 16297,0: POKE  - 16302,0
    : POKE  - 16300,0
130  FOR S = 2 TO NP
140  READ F$
150  PRINT  CHR$ (4)"BLOAD"F$",A"8192 * P
160  GET GO$: PRINT
170  POKE ML - 3 + P,0
180 P = 3 - P
190  NEXT S
200  GET GO$
210  TEXT
220  HOME : PRINT "SHOW OVER."
230  END
240  DATA  "POLAR DESIGN I","POLAR DESIGN II"
```

on the disk is that fewer program lines are required to display a picture than are needed to create the picture within the program. Memory usage is often a concern when developing programs which use high-resolution graphics. The memory segment allocated for storing variables often overlaps with the high-resolution graphics pages. Variables are stored in the memory segment immediately above the program. If the program is lengthy, some of the variables could be stored in the high-resolution graphics memory segments. In act, a very long program can easily extend into the graphics-page area itself.

If either the program or the variable extend into the high-resolution graphics are of memory, adjustments must be made. Problems of this nature are easily detected: if part of a program is erased after a high-resolution graphics picture has been displayed, the program obviously extends into the graphics memory. The problem can generally be solved by using page 2 for displaying all pictures. If the problem persists even when only page 2 is used, then the size of the program must be reduced. If the program cannot be shortened, it must be carefully divided into two or more parts which must then be run one at a time—provided each part is independent of the others. The program parts are independent if none of the variable values need to be passed to the other parts for them to function properly. If any program part is dependent on any other part, then the parts must be chained using the utility program CHAIN on the System Master disk. If the program is intact after a picture has been displayed, but variable values are lost or unexpected lines ap-

pear in the graphics picture, then the program can be solved by using the statement

```
LOMEM :24576
```

in the first line of the program. This causes the program to store its variable values in the memory located above the page 2 high-resolution graphics memory.

High-resolution plot graphics are suitable for any application where picture detail is important. There are no graphics modes available that have a higher screen resolution. There are other methods for forming high-resolution displays, however, ones that place the pictures on the screen more quickly. Vector and raster graphics both create displays on the high-resolution screens. We will discuss these in subsequent chapters.

4

Vector Graphics

This chapter presents the procedures for developing shape tables. A *shape table* is a collection of designs or shapes using high-resolution graphics. A shape or design can be a picture of a rose, circle, face, leaf, or any other figure that needs to be illustrated. Shape tables make programs come to life.

Once a design has been stored in a shape table, it can be easily displayed on the screen, moved around, rotated, or enlarged from an Applesoft program. This flexibility is achieved through the use of vector graphics. Instead of stating particular points on the screen that are to be lighted, this graphics mode names a set of points. The relative positions of the points are described by their distance and direction from a designated point in the set. Descriptions expressed in terms of distance and direction are called *vectors*; graphics that are formed using this technique are called *vector graphics*.

There are many commercial software packages that can help computer users develop and use shape tables. In fact, Chapter 6 contains listings of programs that assist in the construction and use of these devices. The programs are more useful, however, after the user understands the processes they perform. Consequently we recommend that the material presented in this chapter be studied thoroughly before using the Table Developer, Shapely Scenes, and Table Merger programs listed in Chapter 6.

Several steps are required to create a shape table. This chapter explains the complete construction process by discussing designing shapes, constructing a shape table, and entering and saving a shape table.

DESIGNING SHAPES

The initial step in learning to create a table is to examine the procedures for designing a single shape. It is essential to understand that the design itself is not stored in memory or on a disk. What is stored instead is a set of vector instructions called a *shape definition,* which directs the computer to light up certain dots on its display monitor. The instructions have the effect of describing a path on the TV

screen or monitor. Imagine an "electronic pencil" following an imaginary path and lighting up a high-resolution graphics dot whenever the instructions indicate. Whenever the shape is to be drawn on the screen, an Applesoft command is used to instruct the computer to display the shape. The instructions of the shape definition then take over until the shape is completed. The Applesoft commands used in conjunction with shape tables are discussed in Chapter 5.

The following steps are required to define an individual shape or design:

1. Shading a selected design or shape.
2. Tracing the vector path.
3. Coding the vectors.
4. Charting the codes.
5. Coding the chart contents.

Each of the steps is illustrated using a simple triangular shape. We recommend that the procedures presented be duplicated for the triangular shape before other shapes or designs are attempted.

SHADING A SELECTED DESIGN OR SHAPE

Each design or shape that is to be included in a shape table should be sketched or shaded on graph paper before attempting to encode its shape definition. The graph paper used for illustrations in this chapter has been scaled to reflect the height-to-width proportion of the dots on the display screen, which are generally in the ratio of 4:5; regular graph paper is usually adequate.

To draw the shape on graph paper, simply shade the grids needed to form the desired design. Figure 4-1 illustrates a triangular shape shaded on graph paper. Each grid on the graph paper should be regarded as one screen pixel or dot. The lines on the graph paper correspond to the zones between the dots on the screen and, therefore, cannot be lighted.

Shading should be done lightly. Once the grids which form the desired shape have been shaded, the vector path which the computer uses to display the design must be traced. If the shading is not done lightly, the vectors may be difficult to see.

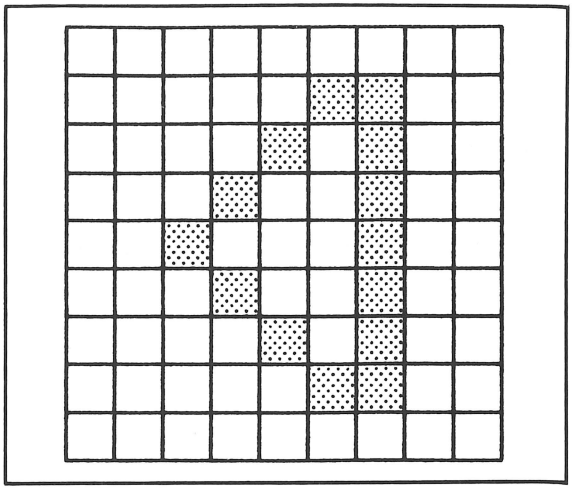


Fig. 4-1. For best results, designs for vector graphics displays should be shaded on graph paper.

TRACING THE VECTOR PATH

The imaginary pencil described earlier can move up, down, right, or left on the screen, but it can only move one dot at a time. The path it is to follow must be described as if it were a dot-to-dot picture. At each dot the pencil stops and is instructed to either light the dot or leave it off.

The path may be started at any point on the screen. Usually it is helpful to start at a point on or in the figure. One important consideration in selecting a starting point is that the Applesoft language allows a defined shape to be rotated, with the starting point as the pivot. Thus the starting point should be selected so that desired visual effects such as spinning, rocking to-and-fro, etc., can be simulated. A more detailed discussion on rotating shapes is presented in Chapter 5.

A map describing the design can now be established. Place a dot in the grid selected as the starting point and trace a path for the electronic pencil to follow. One tracing method that works well is to draw vectors (→) from the starting point to an adjoining grid, then from that grid to another adjoining grid. Continue until a vector points away from each shaded grid in the design. The starting point selected in Fig. 4-2 is the center of the design. The vectors form a grid-to-grid path from the starting point around the design until each shaded

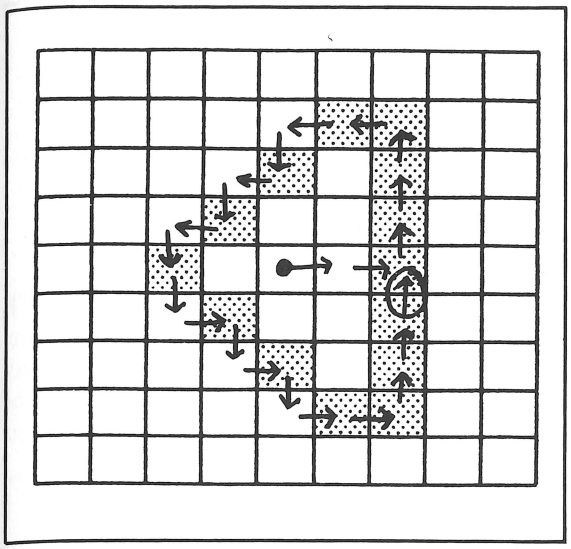


Fig. 4-2. The vector path for a shape is formed by placing vectors end-to-end until a vector points from each shaded grid. The first grid is marked with a dot, and the last vector of the path is circled.

grid has an arrow pointing from it. A good practice is to circle the final vector.

There are guidelines that must be observed for tracing vector paths. Turns must be at 90 or 180 degree angles, as shown in the first line of Fig. 4-3. Vectors that point upward and are outside the shaded grids must follow restrictive conditions of their own. Upward vectors cannot depart from three consecutive unshaded grids of the shape (Fig. 4-3, second line), and upward vectors emanating from two consecutive grids must be followed by a right vector from an unshaded grid or a left vector from an unshaded grid (see Fig. 4-3, third line). The best policy to follow when tracing unshaded regions of a shape is to avoid pointing up more than one grid at a time. These restrictions are rarely a problem, but it is necessary to be aware that they exist.

CODING THE VECTORS

The next step is to make the transition from artist to mathematician. The vector path must be described in terms the computer can understand. Each of the vectors drawn needs to be assigned a three-digit numerical code. If we refer to the point of the vector as its head and to the opposite end as its tail, then two questions must be answered by the vector code.

1. Is the tail of the vector in a shaded grid?
2. In which direction does the head of the vector point?

The code for answering the first question is 0 (meaning no) or 1 (meaning yes). The code for answering the second question is 00 for up, 01 for right, 10 for down, and 11 for left. These codes are easy to remember if they are thought of as the base-2 notations for the numbers 0, 1, 2, and 3. By extending each of the vectors in the appropriate direction from a central point, they can be numbered sequentially in a clockwise direction (see Fig. 4-4).

A typical three-digit vector code is 011, meaning the tail is not in a shaded grid and the vector points to the left. This tells the imaginary electronic pencil not to light the dot containing the tail of the vector and to move to the left one dot on the screen. Another example of a vector code is 110. This code tells the computer to turn on the light and then move down. The first digit corresponds to the status of the grid in which the tail is located (shaded or not shaded) and the last two digits indicate the direction the head of the vector points.

Always code the vector with its tail in the starting grid first. Follow the path established for the design and code each vector in the path. Make a list of the vectors in the order they appear. Figure 4-5 shows the vector code list and a description of each code for the sample triangular shape.

CHARTING THE CODES

The vector codes for each shape must be placed in an eight-column chart. Number the columns of the chart from left to right beginning with 7 and descending to 0. Columns labeled 0, 1, and 2 form the first section, columns labeled 3, 4, and 5 form the second section, and columns labeled 6 and 7 form the third section of the vector code chart. (See Fig. 4-6.)

Place the three-digit codes from the vector code list into the chart, working from right to left.

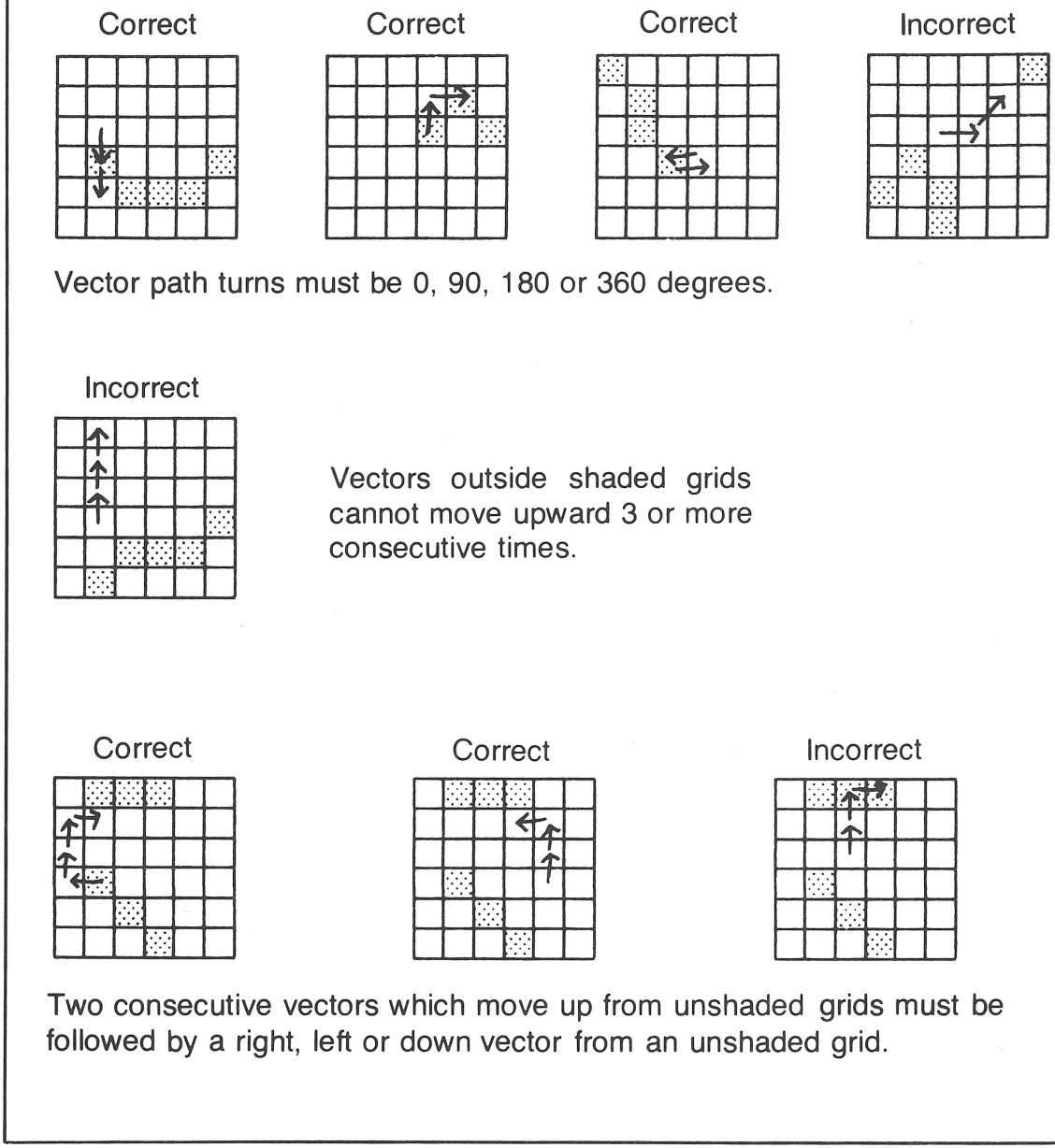


Fig. 4-3. The guidelines depicted in the figure must be followed in forming shape vector paths.

The first vector code must be placed in section one of the top row, as illustrated in Fig. 4-7.

Place the remaining vector codes from the list into the chart using the following guidelines:

1. Section three of the chart can accommodate only the vector codes 010, 011, and 001. Only the trailing two digits can be placed in the section. The leading digit,

always a zero, is omitted from the chart, but is understood by the computer.

2. If any section contains the vector code 0000 (tail in unshaded grid, point up), a section to its left in the same row must contain a vector code having the digit 1 (i.e., 010, 110, 011, 001, 101, 100, or 111).

These guidelines occasionally prohibit the placement of a code in the next vacant section of the chart; it is sometimes necessary to fill the remaining columns of that row with zeros. Some vector code sequences require consideration of both guidelines to insure proper placement in the chart. Correct charting for such a sequence and examples of common charting errors are illustrated in Fig. 4-8.

The chart for the sample triangular shape is presented in Fig. 4-9. Every row on the chart must have a digit in each of its columns. Note that the example contains only zeros in section 3 of the chart; this may not always be the case. If all the vector codes have been used and the row is not completely filled, place a zero in each of the remaining columns. If a row exists in the chart that has zeros in every column, either a coding error has been made or an illegal path has been traced.

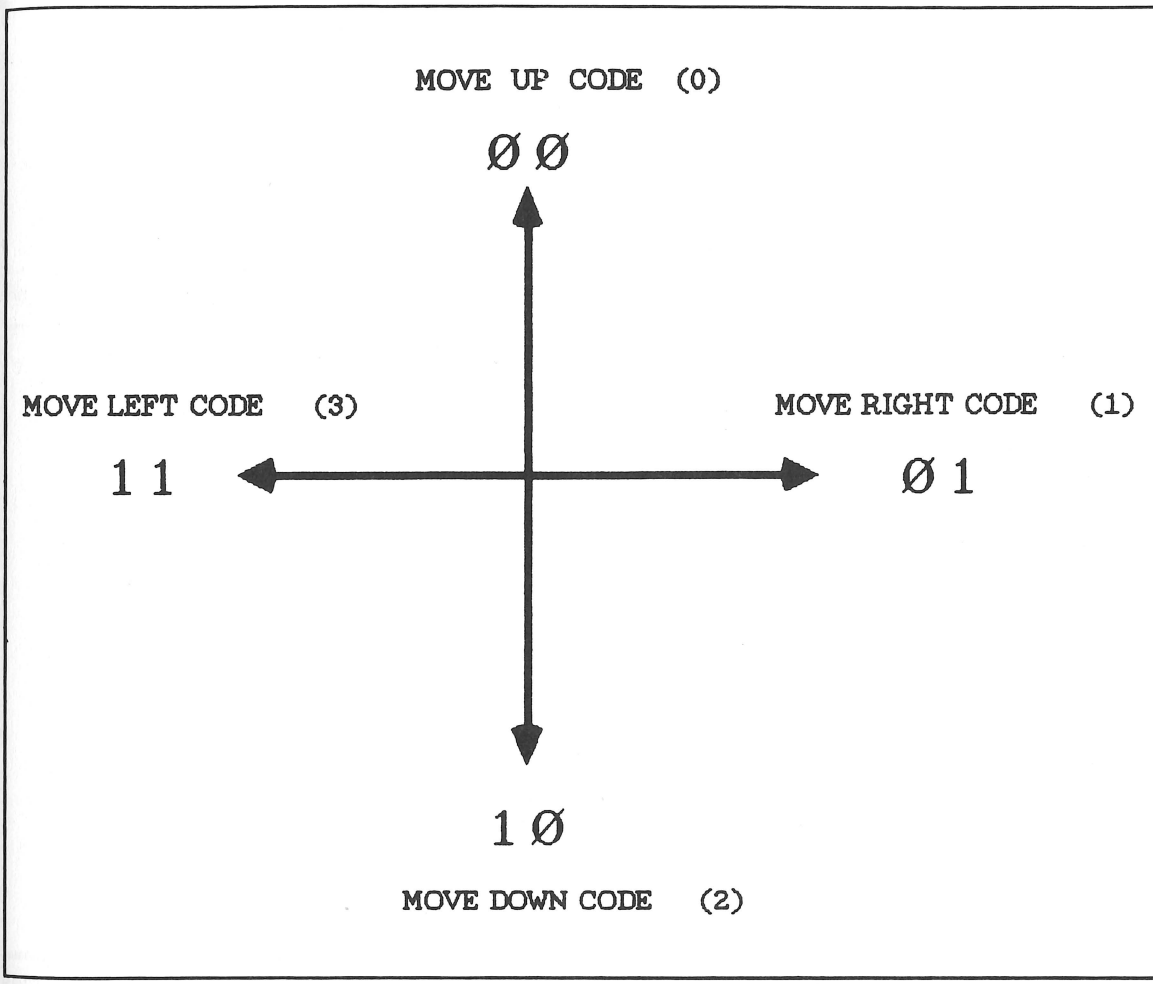
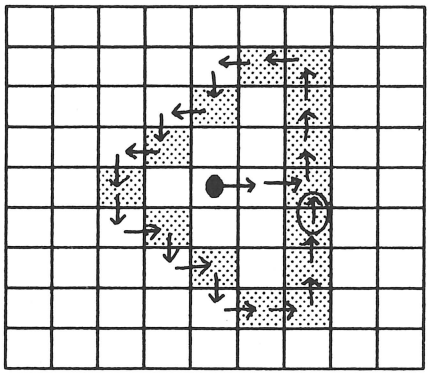


Fig. 4-4. Each vector in the shape vector path is assigned a two-digit binary code which indicates the direction the vector points.



PATH POSITION	VECTOR CODE	CODE MEANING
1	0 0 1	NO PLOT, MOVE RIGHT
2	0 0 1	NO PLOT, MOVE RIGHT
3	1 0 0	PLOT, MOVE UP
4	1 0 0	PLOT, MOVE UP
5	1 0 0	PLOT, MOVE UP
6	1 1 1	PLOT, MOVE LEFT
7	1 1 1	PLOT, MOVE LEFT
8	0 1 0	NO PLOT, MOVE DOWN
9	1 1 1	PLOT, MOVE LEFT
10	0 1 0	NO PLOT, MOVE DOWN
11	1 1 1	PLOT, MOVE LEFT
12	0 1 0	NO PLOT, MOVE DOWN
13	1 1 0	PLOT, MOVE DOWN
14	0 0 1	NO PLOT, MOVE RIGHT
15	1 1 0	PLOT, MOVE DOWN
16	0 0 1	NO PLOT, MOVE RIGHT
17	1 1 0	PLOT, MOVE DOWN
18	0 0 1	NO PLOT, MOVE RIGHT
19	1 0 1	PLOT, MOVE RIGHT
20	1 0 0	PLOT, MOVE UP
21	1 0 0	PLOT, MOVE UP
22	1 0 0	PLOT, MOVE UP

Fig. 4-5. This figure gives the meanings of all the vector codes for the triangle shape. The first digit of the code indicates whether the vector points from a shaded or unshaded grid. The second two digits indicate the direction the vector points.

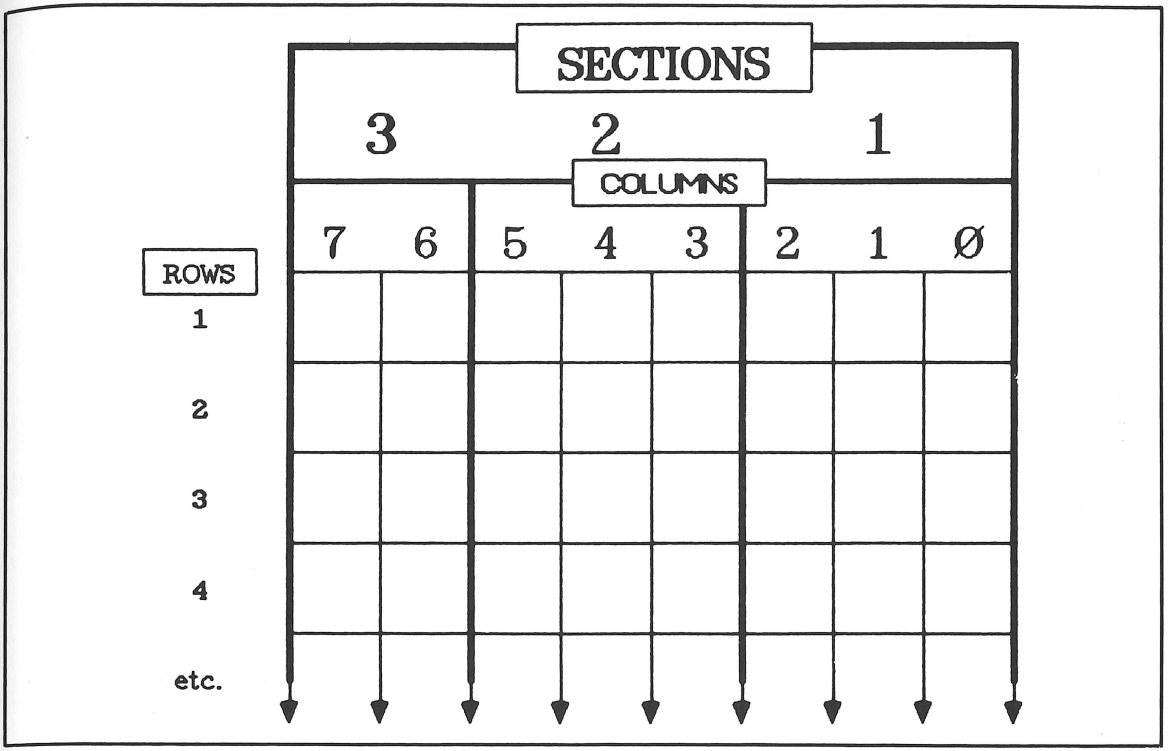


Fig. 4-6. Vector codes charts are used to arrange the vector codes in a meaningful way.

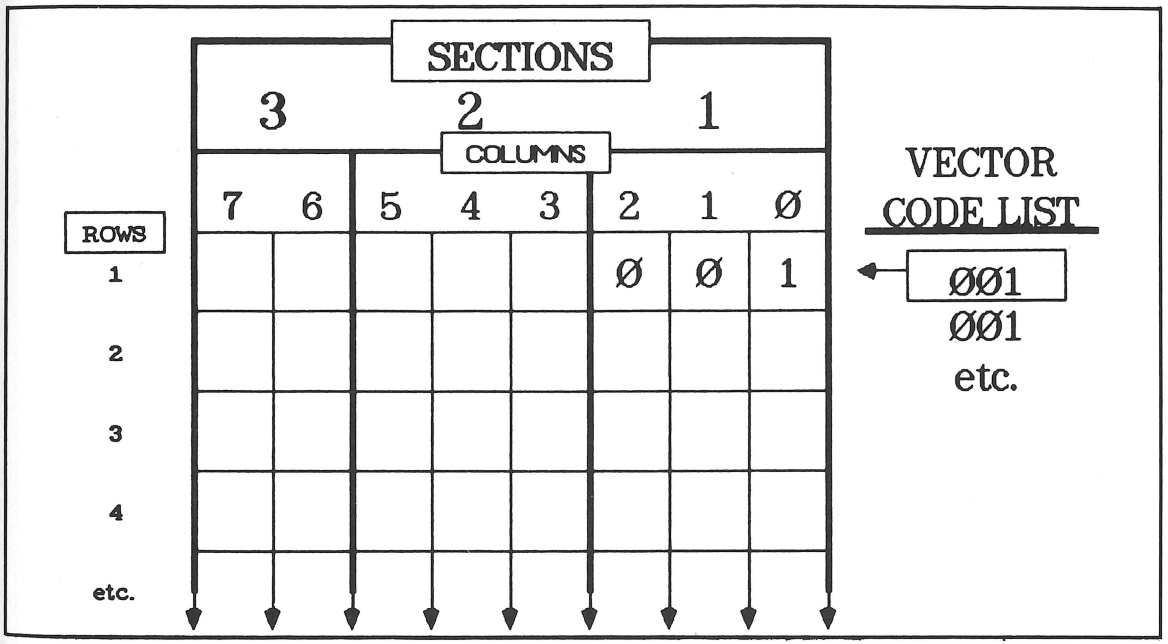
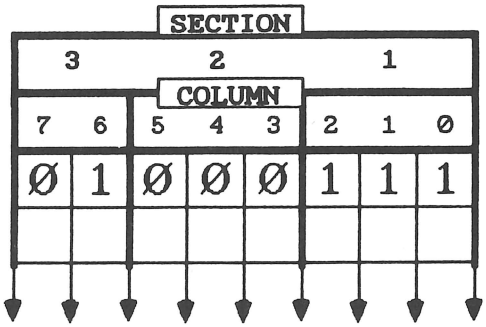


Fig. 4-7. The codes that describe the vector path of a shape are placed in rows of the vector code chart, starting in section 1 of the chart.

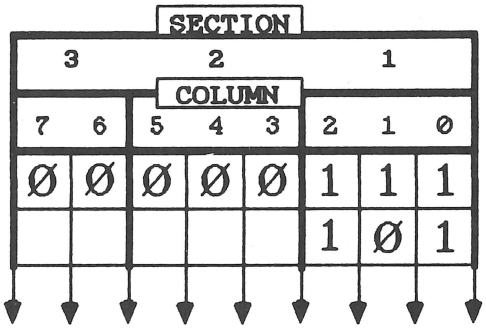
VECTOR CODE LIST

1 1 1
Ø Ø Ø
1 Ø 1



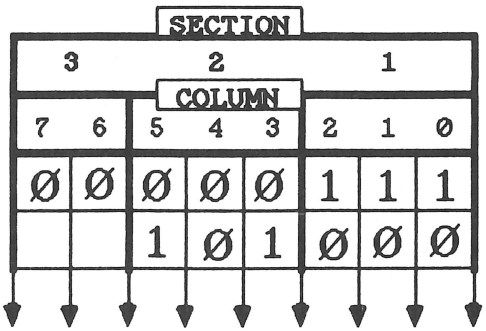
Incorrect

Violates guideline 1. The code 101 is not allowed in Section 3 of the chart.



Incorrect

Violates guideline 2. The code 000 in Section 2 will be ignored since no column to its left contains the digit 1.



Correct

Satisfies both guidelines.

Fig. 4-8. Care must be taken to follow specific guidelines for placing the vector codes in the chart.

ROWS	SECTIONS 3		2			1		
	COLUMNS 7	6	5	4	3	2	1	Ø
1	Ø	Ø	Ø	Ø	1	Ø	Ø	1
2	Ø	Ø	1	Ø	Ø	1	Ø	Ø
3	Ø	Ø	1	1	1	1	Ø	Ø
4	Ø	Ø	Ø	1	Ø	1	1	1
5	Ø	Ø	Ø	1	Ø	1	1	1
6	Ø	Ø	Ø	1	Ø	1	1	1
7	Ø	Ø	Ø	Ø	1	1	1	Ø
8	Ø	Ø	Ø	Ø	1	1	1	Ø
9	Ø	Ø	Ø	Ø	1	1	1	Ø
10	Ø	Ø	1	Ø	Ø	1	Ø	1
11	Ø	Ø	1	Ø	Ø	1	Ø	Ø

VECTOR CODE LIST	ROW	SECTION
001	1	1
001	1	2
100	2	1
100	2	2
111	3	1
111	3	2
010	4	1
111	4	2
010	5	1
111	5	2
010	6	1
110	6	2
001	7	1
110	7	2
001	8	1
110	8	2
001	9	1
110	9	2
101	10	1
100	10	2
100	11	1
100	11	2

Fig. 4-9. This figure shows the proper placement of the triangle vector codes in the chart.

CODING THE CHART CONTENTS

Only one step remains in the coding process. Darken the line between columns 3 and 4 of the chart. This line divides the chart into two new sections of four columns each (see Fig. 4-10). Each row now contains two four-digit binary numbers. These numbers must be translated to their hexadecimal (hex) equivalents. The two hex digits thus obtained for each row describe the vector codes in terms the computer can understand. Either the Base Conversion program listed in Appendix B or the information in Table 4-1 can be used to convert the binary numbers to their hex equivalents. The binary-to-hex conversion for the sample shape is presented in Fig. 4-10.

After the conversion from binary to hex is completed, append the number 00 to the hex code list. The 00 at the end of the list is a signal to the computer that the shape definition is finished.

The list formed by the hex numbers is the *shape definition,* or set of instructions, used by the computer to draw the design on the screen. The coding procedures are time-consuming at first, but speed increases as the codes become more familiar. After coding two or three shape definitions, coding time can be decreased by placing the vectors for the path of a shape directly into the vector chart, without

SECTIONS

	3		2			1			
ROWS	COLUMNS 7	6	5	4	3	2	1	Ø	HEX CODES
1	Ø	Ø	Ø	Ø	1	Ø	Ø	1	→ 09
2	Ø	Ø	1	Ø	Ø	1	Ø	Ø	→ 24
3	Ø	Ø	1	1	1	1	Ø	Ø	→ 3C
4	Ø	Ø	Ø	1	Ø	1	1	1	→ 17
5	Ø	Ø	Ø	1	Ø	1	1	1	→ 17
6	Ø	Ø	Ø	1	Ø	1	1	1	→ 17
7	Ø	Ø	Ø	Ø	1	1	1	Ø	→ 0E
8	Ø	Ø	Ø	Ø	1	1	1	Ø	→ 0E
9	Ø	Ø	Ø	Ø	1	1	1	Ø	→ 0E
10	Ø	Ø	1	Ø	Ø	1	Ø	1	→ 25
11	Ø	Ø	1	Ø	Ø	1	Ø	Ø	→ 24

ØØ - SIGNALS END OF SHAPE

Fig. 4-10. After the vector codes have been placed in the chart, each row of the chart represents an eight-digit binary number that must be translated to hexadecimal form.

Table 4-1. Base Conversion Chart.

Binary	Hexadecimal	Decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

compiling the vector code list.

The complete shape definition is one element of a shape table. The remaining elements are discussed in the following section.

CONSTRUCTING A SHAPE TABLE

Up to this point, instructions have focused on defining a specific shape. This skill is merely one facet of the overall task of creating a complete shape table. This section explains how to select shapes for a table, stack the shape definitions, and indexing the table. Then we will look at the completed table.

Selecting Shapes for a Table. As many as 255 designs or shapes may be stored in a shape table. Since most programs will not require 255 different designs, the shapes for several programs can be stored in a single table. If a program is quite long, tables with fewer entries may be used to conserve memory.

Another point to consider while planning a shape table is that shapes may be placed on the screen using different colors, but each shape can be only one color. If a white house with a blue roof and orange shutters is to be drawn, separate shape definitions for each part of the house must be defined in the table.

Designs selected to illustrate the steps required to assemble a shape table are pictured in Fig. 4-11. The charting and coding information for each shape definition is located in Appendix C.

Stacking the shape definitions. Shading, coding and conversion to hex procedures must be completed for each design that is to be placed in a shape table. The hex codes for each shape definition are then consolidated into a single list (see Fig. 4-12). The code list for the sample shape table was formed by stacking the shape definitions in the following order: a triangular design, a half note, a treble clef, a map of Nebraska, and a maple leaf.

Indexing the table. Since the sample shape table contains several shape definitions, an index of the shape table must be created. All shape tables need an index even those consisting of only one shape definition. A *shape table index* has two functions: it states the number of shape definitions in the shape table and the relative memory locations of those shapes within the table. Each entry of the index must be expressed as a 4-digit hex number. However, since each byte of computer memory can only store a 2-digit hex number, it is necessary to split each 4-digit number into two parts. Furthermore, due to the way the computer reads information from its memory, the order of the 2-digit numbers must be reversed. For example, a 4-digit hex number like 0005 must be split into two codes 00 and 05; then the order of the codes is reversed so that they appear in the index as 05 00.

The index for the sample shape table must contain six 4-digit numbers or codes. The first code represents the number of shapes in the table, 0005, and the remaining five represent the "distance," measured as a number of memory bytes, from the start of the shape table to each of the shape definitions. These codes are more easily determined after the shape table has been entered in the computer, so a temporary code of 0000 is used for each shape's relative address. The complete temporary index for the sample shape table is:

05 00 00 00 00 00 00 00 00 00 00 00

Naturally, the index needs to be corrected after the shape table has been entered into the computer.

The completed shape table. The temporary index should be placed at the beginning of the shape table code list. Sections of the list should appear in

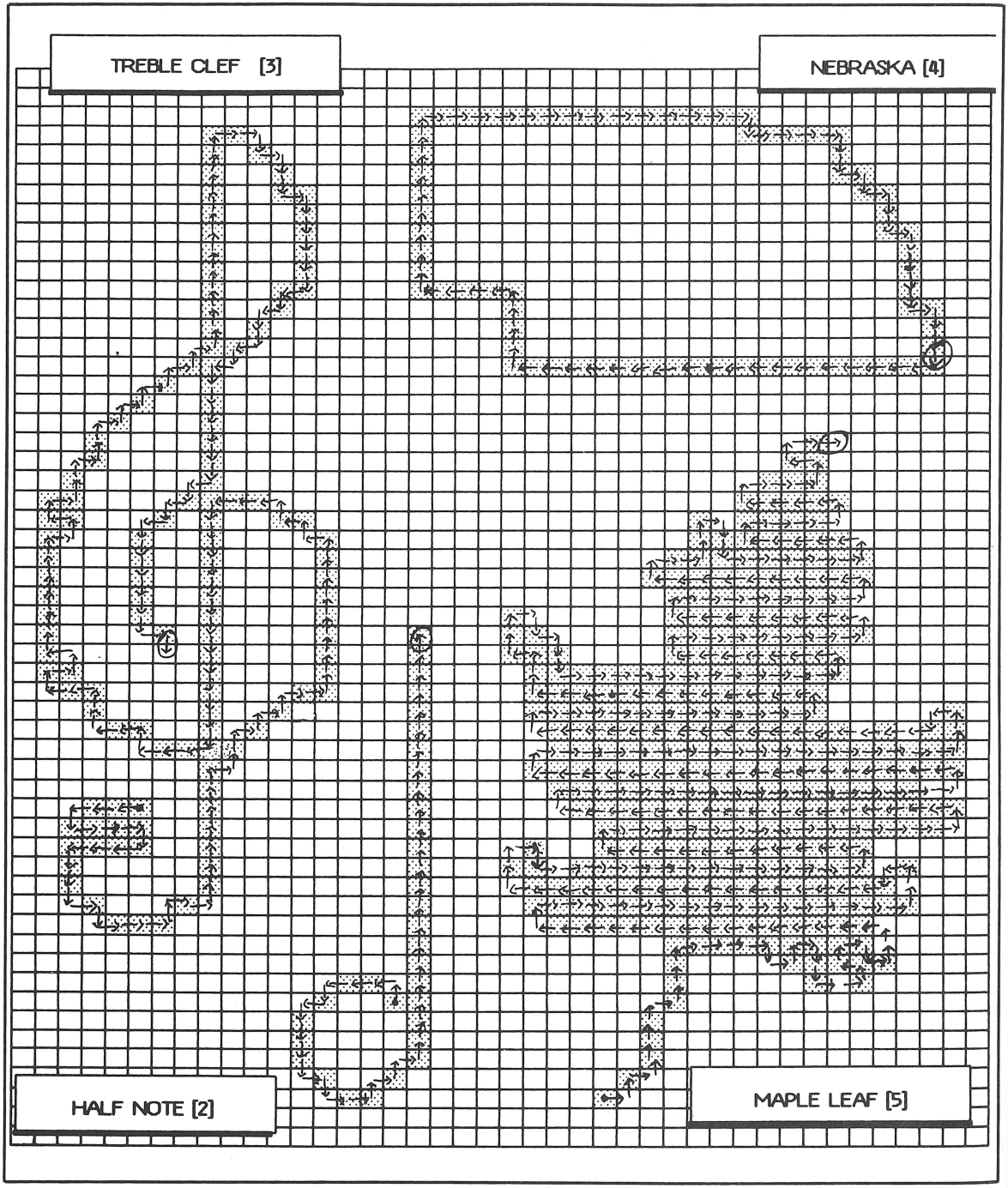


Fig. 4-11. This figure shows the other four shapes that are placed in the Sample Shapes shape table with the triangle. The position of each shape in the table is noted in the figure.

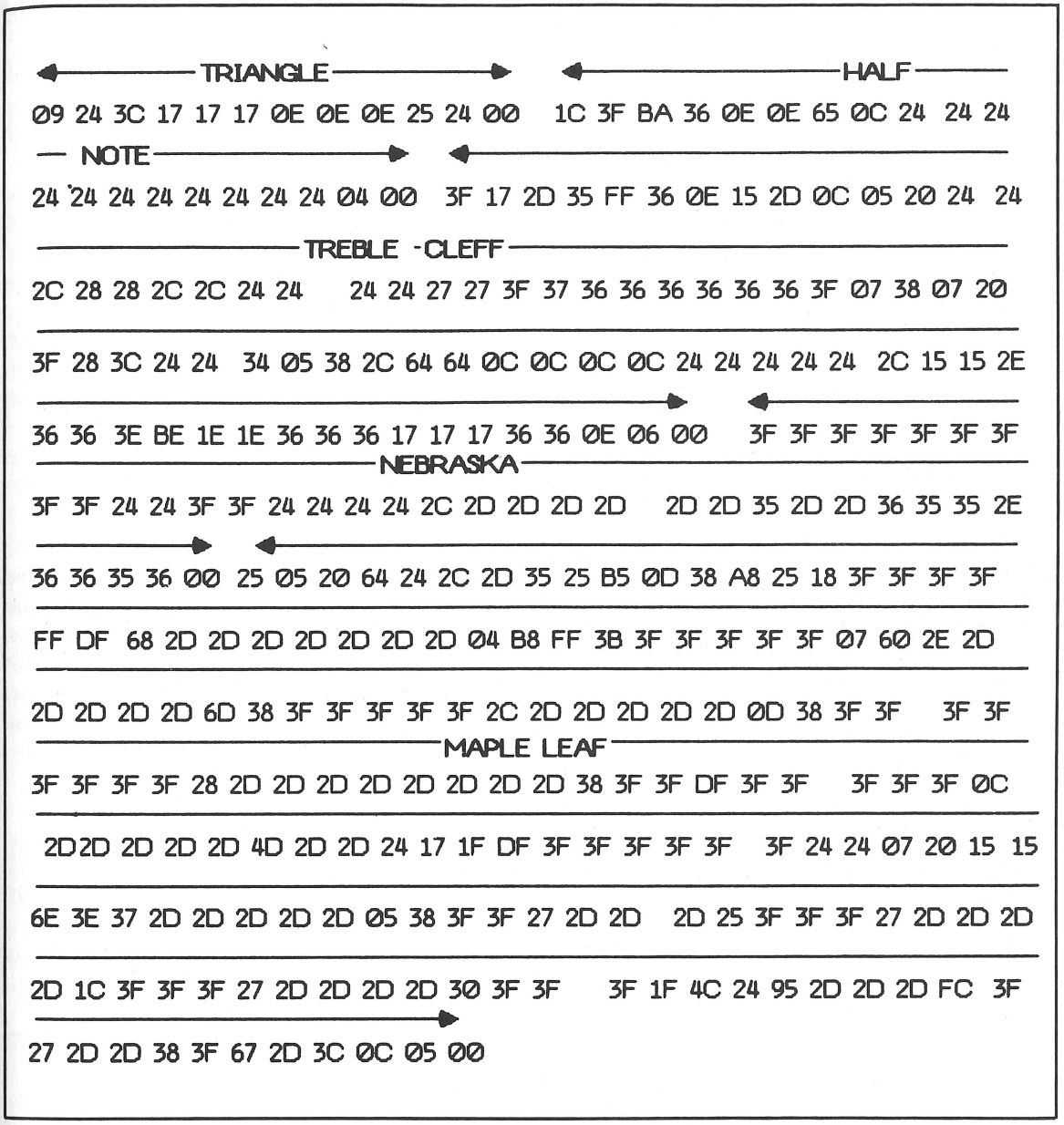


Fig. 4-12. The hex codes of all the shapes that are to be placed in the shape table are consolidated into a single list of codes.

this order: index, triangle shape definition, note shape definition, treble clef shape definition, Nebraska shape definition, and maple leaf shape definition. When the procedures described have been completed, the information from the table can be entered into the computer.

ENTERING AND SAVING A SHAPE TABLE

The designs have been charted and coded. Now it is time to supply the computer with the

information it needs to display the shapes on the screen. This section describes how to enter the shape table, correct the index, and save the table on disk.

Entering the Shape Table. The completed sample shape table has 330 codes (14A hex). the number of codes in a shape table is an important factor in deciding where the table should be placed in memory. Guidelines for selecting appropriate locations in memory to place shape tables are discussed in Chapter 5.

Although shape tables can be placed in the computer's memory from Applesoft BASIC programs using POKE statements, the easiest and fastest way to enter one is to access the Apple Monitor and type the list of codes directly into memory. This is the approach we discuss in this book.

With the shape table code list, this book, and an initialized disk in hand, proceed to an available computer. After booting the DOS, type CALL – 151 and press the Return key. This causes the computer to invoke the Apple Monitor, which displays an asterisk prompt character. From here on, all references to memory addresses will be in hexadecimal; it is customary to use the dollar sign ($) before a hex number.

The memory address selected for the initial code of the shape table is $3000. Each address can store exactly one hexadecimal code, so we will store the 2-digit hex codes of the shape table at successive addresses in memory. The first code of the sample shape table (05) is stored in address $3000; the second code, 00, in $3001; the third code, 00, in $3002, etc.

In order to enter the shape table, all you need do is type the starting address of the table next to the asterisk, followed immediately by a colon and then the hex codes of the table separated by spaces. The Apple Monitor will allow up to 85 codes to be entered into successive addresses in memory merely by placing a space between the codes. This means that approximately six rows of the screen can be filled without pressing the Return key.

The first two segments of code, including the index and triangular shape definition, are listed in Listing 4-1. Do not press the Return key until all the codes in the first segment have been entered.

The second and subsequent segments of code may be entered by typing a colon next to the asterisk, followed by the next section of code.

The process should be continued until all the codes in the shape table have been entered. After the final code has been typed into memory, the Return key should be pressed to complete the procedure. In order to check the information in memory to see if it corresponds to the shape table list, type 3000 beside the prompt character (*) and press the Return key. The computer should respond with

```
3000 – 05
```

To view more of the code list, press Return again to see the display.

```
3001 – 00 0C 00 18 00.
```

Listing 4-1. Entering a shape table into memory.

```
*3000:05 00 00 00 00 00 00 00 00 00 00 00
 09 24 3C 17 17 0E 0E 0E 25 24 00 1C 3F
 BA 36 0E 65 0C 24 24 24 24 24 24 24 24
 24 24 24 04 00
*:3F 17 2D 35 FF 36 0C 15 2D 0C 05 20 24
 24 2C 28 28 2C 2C 24 24 24 24 27 27 3F
 37 26 26 26 26 26 26 2F 07 38 07 20 3F
 etc.
```

Continue pressing Return until the entire list of code has been displayed. Caution: a single error in the shape table may cause it to function improperly. Make sure the entries are correct now to save a lot of additional work later.

If one of the codes is incorrect, it can be corrected by typing the address of the byte containing the errant code, followed by a colon and then the correct code. If one or more of the codes as been omitted, the problem is a little more difficult to correct. One remedy is to retype all the codes from the omission to the end of the shape table, but a better solution is to use the Monitor "move" command to relocate the portion of the shape table beyond the omitted code in another area of memory, type in the missing code, and then move the relocated portion of the table to its proper position. Techniques for editing shape tables are discussed in more detail in Chapter 5.

Correcting the Index. Once the sample shape table has been entered, the index can be corrected easily. The starting address of each shape definition must be determined. This is most easily accomplished by listing the shape table codes. Type

```
3000.3050
```

next to the Monitor prompt character, and the contents of all the bytes in the memory range $3000 to $3050 will be displayed on the screen. The address of the first byte of each shape definition can be determined by looking at the codes listed on the screen.

Recall that each shape definition ends with the code 00. The code immediately following a 00 code is the first code of a shape definition. The address which contains the first code of a shape definition is needed for completing the shape table's index. Jot down the starting address of each shape definition in the range $3000 to $3050, then type 3050.3100. Note the addresses of shape definitions that begin in this range, then display the range $3100 to $3150 to find the starting addresses of the remaining shape definitions.

The starting addresses of the shape definitions in the sample shape table are: triangle, $300C; note $3018; clef, $302D; Nebraska, $307C; and maple leaf, $30A0. The relative address (the "offset") of each shape definition from the beginning of the shape table can be found by subtracting the absolute address of the first byte of the shape table ($3000) from the absolute address of the first byte of each shape definition in the table. Mathematical operations performed in the base-16 number system require caution and a good knowledge of the number system; the starting address of the sample shape table was selected to make this task easier. The relative address of each shape definition in the table can be determined by replacing the first digit of its absolute address with the number 0. Hence, the 4-digit codes that are needed to complete the index of the sample shape table are 000C, 0018, 002D, 007C, and 00A0.

Recall that each code must be split into two 2-digit codes, and that the order of the 2-digit codes must be reversed. The correct codes for the index can be placed in the table by typing the following line after the Monitor prompt character:

```
3002:00 0C 00 18 00 2D 00 7C 00 A0 00.
```

Once the index has been corrected the completed shape table is ready for use, but it is a good idea to save the shape table on disk before trying to display the designs.

Saving the Table. The sample shape table can now be saved on a disk. First exit the Monitor by typing CTRL-C and then Return. Make sure a formatted disk is in the disk drive and type

```
BSAVE SAMPLE SHAPES, A$3000, L$14A
```

then press Return.

The BSAVE command indicates that the file to be saved is in machine language form (binary code). Sample Shapes is the name selected to identify the shape table's file on the disk, A$3000 indicates the hex address in memory where the shape table begins, and L$14A indicates the length of the shape table in hex notation.

Techniques for using shape tables in BASIC programs are presented in Chapter 5.

5

Using Shape Tables In BASIC Programs

The shape table developed in Chapter 4 is used throughout this chapter to illustrate the use of BASIC commands related to shape tables. You should enter and save it on a disk before studying this chapter, where we will discuss the following topics:

1. Displaying shapes on the screen.
2. Shape table related commands.
3. Demonstration programs.
4. Mixing text and graphics
5. Ideas for programs.
6. Helpful hints.

DISPLAYING SHAPES ON THE SCREEN

To test the sample shape table, enter and run the Debut program (Listing 5-1). A discussion of the BASIC commands used in the program is presented following the program listing.

After entering the Debut program, type RUN, and press Return. If the shapes displayed are satisfactory, then the shape table was created successfully. If no shapes appear on the screen or their appearances are distorted, refer to the Helpful Hints section of this chapter for debugging clues and editing instructions.

SHAPE TABLE RELATED COMMANDS

The Applesoft BASIC language has several special commands that allow nice things to be done with shape tables. Statements unique to shape table utilization are DRAW, XDRAW, ROT, and SCALE; other graphics statements needed for shape table utilization are HGR, HGR2, HPLOT, and POKE. Since most of these statements appear in the Debut program, a discussion of the program is presented to illustrate the commands pertinent to shape table programming.

Listing 5-1. Debut program.

```
5  REM **  DEBUT PROGRAM  **
10  HOME
15  POKE 232,0: POKE 233,96: REM  POKES ADDRESS OF TAB
     LE INTO MEMORY
20  PRINT  CHR$ (4)"BLOAD SAMPLE SHAPES,A$6000": REM
     LOAD TABLE FROM DISK
30  HGR2 :C = 0: REM  SETS GRAPHICS MODE, PAGE 2
40  HCOLOR= 3: REM  SETS COLOR FOR DRAWING SHAPES TO W
     HITE
50  SCALE= 1: REM  SHAPE DRAWN AT DEFINED SIZE
60  ROT= 0: REM  SHAPE DRAWN WITHOUT ROTATION
70  FOR X = 1 TO 5: REM  LOOP DRAWS ALL 5 SHAPES ON SC
     REEN
80 C = C + 40: REM    CHANGES COLUMN COORDINATES FOR D
     RAWING EACH SHAPE
90  DRAW X AT C,100: REM  DETERMINES SHAPE TO BE DRAWN
      AND WHERE
100  NEXT X
110  FOR X = 1 TO 5000: NEXT X: REM  LOOP FOR VIEWING
     PLEASURE
120  HGR2 : HOME : TEXT : REM  CLEARS GRAPHICS AND TEXT
      PAGES, RETURNS TO TEXT
130  INPUT "WANT TO VIEW SHAPES AGAIN? (Y/N) ";G$
140  IF G$ = "Y" THEN 30: REM  REPEATED VIEWING OPTION
150  END
```

Lines 10-15: After clearing the text screen with the HOME statement, use POKE statements to tell the computer where the shape table is stored in memory. When the Apple tries to display a shape from a shape table, it first looks into memory locations $00E8 and $00E9 to find the address of the first code of the shape table. The sample shape table's first code is stored in address $3000, so it is necessary for that information to be provided before the computer can display the designs. The last two hex digits of the address (00) must be stored in location $00E8, and the first two hex digits (30) of the address must be stored in location $00E9 (see Fig. 5-1).

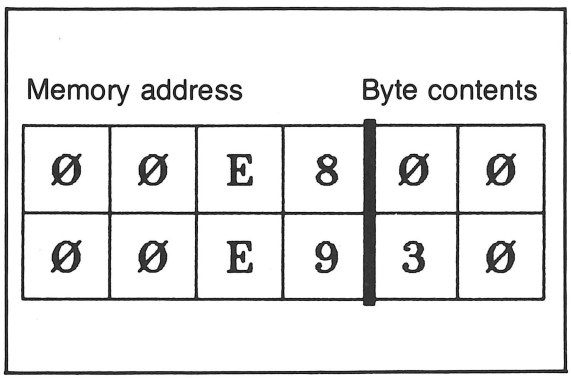


Fig. 5-1. The starting address of the shape table must be placed in memory locations $E8 and $E9 before the table can be used in a BASIC program.

Line 15 of the Debut program places the shape table's starting address into the proper memory locations. In order to do so from a BASIC program, the POKE command is needed. POKE statements use decimal values instead of hex values, so the hex numbers E8, E9, 30, and 00 mentioned above need to be converted to decimal numbers: 232, 233, 48, and 0, respectively. The statement POKE 232,0 places the nex number 00 into memory location $E8, while POKE 233,48 places the hex value 30 into memory location $E9.

If line 15 of the Debut program is omitted, the desired shapes do not appear on the screen. Depending on the contents of memory locations 232 and 233, the Apple either draws some very strange images (not from the shape table), draws nothing at all, hangs up (press CTRL-RESET to recover), gives an error message that cannot be seen because the graphics page is in the way, or jumps into the Apple Monitor. In essence, it is absolutely necessary to include the POKE statements in the program before trying to use the shape table.

Line 20: Line 20 of the Debut program copies the shape table from the disk and places it in memory. Since this is done from within the program, the format for deferred-mode DOS commands is used. CHR$(4) is the ASCII equivalent to CTRL-D and must precede all DOS commands issued from an Applesoft program. The statement BLOAD SAMPLE SHAPE must be enclosed in quotes. If the shape table is not to be placed back into the memory locations from which it was saved (with BSAVE), the new address at which it is to start must be specified. A discussion of the address parameter is presented in the Helpful Hints section of this chapter.

Line 30: HGR2 causes the Apple to display its high-resolution graphics page 2. Since HGR causes page 1 of high-resolution graphics to be displayed, HGR2 is used in the Debut program because the sample table was placed into memory locations that would be erased by HGR.

The variable C is initialized in line 30 because the program offers the option of viewing the shapes repeatedly. Since C designates the column coordinates where shapes appear on the screen, it needs to be reset each time the repeat option is exercised. This prevents the program from exceeding the 280-dot width of the high-resolution graphics screen.

Line 40: This line sets the high-resolution color used to display the shapes on the screen. HCOLOR=3 causes the shapes to appear white. Any of these eight colors available with high-resolution graphics may be chosen:

0	Black 1	4	Black 2
1	Green	5	Orange
2	Violet	6	Blue
3	White 1	7	White 2.

Since the color 3 is available over the entire screen, it was selected for the Debut program. A detailed discussion of the high-resolution color properties is presented in Chapter 3. The best way to determine the effects of each color is to experiment. A program designed for such experimentation is provided later in this chapter.

Line 50: This line contains the statement SCALE=1. SCALE=1 instructs the computer to draw the shape exactly as it is defined in the shape table. The value used in the statement can be any number from 0 through 255. SCALE=2 causes the shape to be drawn twice its defined size, SCALE=3 results in an image 3 times its defined size, etc. SCALE=0 is used for maximum size. When the value of SCALE is 2, each vector described in the shape definition is used twice; the value 3 results in repeating each vector three times. As the value of SCALE increases, the likelihood of shape distortion also increases. Solid shapes will not be solid when SCALE is greater than 1.

Line 60: The statement ROT=0 instructs the computer to draw the shape on the screen oriented exactly as it was defined. The value of ROT must be between 0 and 255. ROT=16 results in the shape being drawn at a 90-degree clockwise rotation, ROT=32 produces a 180-degree clockwise rotation, ROT=48 results in a 270-degree clockwise rotation, and the cycle starts over with ROT=64. A shape with SCALE=1 can be drawn in eight rotation positions. Any number in the allowable range for

Table 5-1. Shape Rotation Positions vs. Scale Value.

Scale Value	# Rotation Positions
1	8
2	16
3	16
4	16
5	40
6	40
7	48
8	48
9	64
>9	64

ROT may be used, regardless of the SCALE value; however, only certain ROT values are recognized by the computer. If a number is used that does not correspond to an available postion for a particular SCALE value, the computer selects a position. If SCALE=1, the statement ROT=18 will have the same effect as ROT=16 because the value used is the next smaller recognized rotation value. As the value of SCALE increases, the number of rotation positions recognized increases (see Table 5-1).

Lines 90-100: The next shape table related statement appears in line 90. DRAW is the command that causes a shape to be displayed on the screen. The statement DRAW X AT C,R instructs the computer to draw the Xth shape (X=1,2,3,4, or 5) in the shape table starting at the point whose coordinates on the screen are C,100 (C=30,60,90,120, or 150). Therefore, the first shape in the table (the triangle) is drawn starting at column 30, row 100, etc. The legal values for C and R are the same as for the HPLOT command: C from 0 to 279, and R from 0 to 191 when using HGR2 (page 2 of high-resolution graphics).

Lines 110-150: The remainder of the Debut program allows the option of viewing the shapes repeatedly without reloading the shape table from the disk.

Although this program has been explained in detail, there are other commands that can be used with shape tables. For example, a programming statement related to shape table utilization that does not appear in the Debut program is XDRAW. DRAW 1 AT 20,20 results in the first shape definition in the table being drawn starting at column 20, row 20, on the screen. If that statement is followed by XDRAW 1 AT 20,20 the computer erases the shape from that position without erasing the background. Thus the illusions of motion and animation can be achieved through proper utilization of the DRAW and XDRAW commands. A technique for simulating motion is discussed later in this chapter.

DRAW and XDRAW may be used without the coordinate parameters. The statement HPLOT 50,100 followed by the statement DRAW 1 results in the same display as DRAW 1 AT 50,100. In general, DRAW N results in the Nth shape of the shape table being displayed at the last high-resolution graphics coordinates used. If no points are plotted before using this form of either the DRAW or XDRAW commands, no figures are displayed on the screen.

DEMONSTRATION PROGRAMS

To appreciate fully the potential of shape tables, it helps to experiment with various applications. Demonstration programs that can assist your experimentation are provided in this section. The programs were specifically designed to illustrate color combinations, effects of scale and rotation, and simulation of motion.

Color Combinations

The Color Trial program (Listing 5-2) demonstrates the color effects possible in high-resolution graphics. Some color combinations do not produce the expected results. By experimenting, however, combinations of colors suitable for most applications can be found. The purpose of Color Trial is to establish what can and cannot be done with certain colors.

It is important to note that background color, shape color, shape definition number, and the locations where shapes appear on the screen are selected while the program is running. This means that many color combinations may be viewed without modifying the program. Before running the program, insert the disk containing the Sample Shapes file in the disk drive.

Lines 10-30: After the text screen is cleared

Listing 5-2. Color Trial.

```
5  REM **   COLOR TRIAL   **
10  HOME
20  PRINT CHR$(4)"BLOAD SAMPLE SHAPES": REM  COPIES
     TABLE INTO MEMORY FROM DISK
30  POKE 232,0: POKE 233,48: REM  STARTING ADDRESS OF
     TABLE IN MEMORY
40  INPUT "BACKGROUND COLOR (0 TO 7) =  ";BG
50  INPUT "SHAPE COLOR (0 TO 7) =  ";SC
60  INPUT "SHAPE DEFINITION (1 TO 5) =  ";SD
70  INPUT "COLUMN COORDINATE (0 TO 279) =  ";C
80  INPUT "ROW COORDINATE (0 TO 191) =  ";R
90  HGR2 : HCOLOR= BG
100  SCALE= 1: ROT= 0
110  HPLOT 0,0: REM  PLOTS A POINT BEFORE BACKGROUND C
     OLOR CAN BE SET
120  CALL 62454: REM  SETS BACKGROUND TO LAST PLOTTED
     COLOR
130  HCOLOR= SC
140  DRAW SD AT C,R
150  FOR X = 1 TO 3000: NEXT X: REM  TIME LOOP
160  HGR2 : TEXT : HOME
170  INPUT "WANT TO TRY ANOTHER COMBINATION? (Y/N)  ";
     Y$
180  IF Y$ = "Y" THEN 40
190  END
```

in line 10, line 20 loads the Sample Shapes shape table into memory. Since no address parameter is used, the shape table will be placed in memory starting at $3000. Line 30 informs the computer where to find the shape table by placing the starting address in memory locations $E8 and $E9.

Lines 40-50: The background color of the high-resolution screen must be specified by the user when line 40 is executed. The color code selected is assigned to the variable BG; any high-resolution code can be specified. The background color normally used by the computer is black 1, which has the color code 0.

The color that is to be used for displaying the selected shape is obtained in line 50. SC is the variable to which the shape color code is assigned. The shape cannot be seen when the values of BG and SC are equal. No shape is visible when BG is 0 and SC is 4 or vice-versa, or when BG is 3 and SC is 7 or vice-versa. The displayed shape may be blurred when either BG or SC is in the range 0 through 3 and the other is in the range 4 through 7. All other combinations are compatible and result in visually sharp displays.

Line 60: The number of the shape that is to be displayed is obtained in line 60. It is assigned to the variable SD. Since Sample Shapes contains only five shapes, the range of legal values for SD is 1 through 5. Entering any other value elicits an ILLEGAL QUANTITY error from the computer. The treble clef (3), Nebraska (4), or the maple leaf (5) are good shapes to use with the program. They are large enough that shape irregularities caused by the combination of colors specified can be detected easily.

Lines 70-80: The screen point for displaying the selected shape is obtained in lines 70 and 80. The column coordinate of the point is assigned to

the variable C, and the row coordinate of the point is assigned to the variable R. To allow the user to observe the wraparound that occurs when certain shapes are positioned too close to the edge of the screen, the value 5 should be specified for both coordinates when shape number 4 is selected.

Lines 90-100: Line 90 clears and displays high-resolution graphics page 2. The normal scale and rotation values for displaying shapes are set in line 100. The Scale and Rotation program listed later in the chapter is designed to allow you to observe the visual effects of different scale and rotation values. However, you may want to try other values in line 100 so you can observe the effects of scale and rotation with various color combinations.

Lines 110-120: These lines are needed to set the background color of the screen. HCOLOR must be set to the desired background color before the other statements in these lines are executed. The ROM subroutine starting at memory location 62454 causes all the points on the high-resolution graphics screen to be changed to the color of the most recently plotted point; the statement CALL 62454 causes the subroutine to be executed. Line 110 plots a point using the desired background color, so the proper values are set before the subroutine is executed.

Lines 130-150: HCOLOR is set to SC, which is the code for the color that is to be used to display the shape. Line 140 causes the specified shape to be displayed at the designated point on the screen. Line 150 is a time loop to keep the shapes on the screen long enough for the user to view. Some of you may wish to replace the time loop with GET GO$. The change will cause the shape to remain on the screen until a key is pressed.

Lines 160-190: These lines allow you to test as many color combinations as desired without exiting the program. The program ends if you do not press the Y key when asked if you want to try another combination.

Effects of Scale and Rotation

The Scale and Rotation program (Listing 5-3) demonstrates the effects different SCALE and ROT values have on the appearance of shapes. Shapes can be selected while the program is running, so that the effects of SCALE changes on both solid and outlined shapes can be observed. Again, the disk with Sample Shapes must be in the disk drive when the program is run.

Listing 5-3. Scale and Rotation demonstration program.

```
5  REM **  SCALE AND ROTATION  **
10  HOME :D$ =  CHR$ (4): REM  ASSIGN CONTROL D CHARAC
    TER TO VARIABLE D$
20  PRINT D$"BLOAD SAMPLE SHAPES"
30  POKE 232,0: POKE 233,48
40  INPUT "SHAPE DEFINITION  (1 TO 5) =  ";SD
50  INPUT "SCALE VALUE (0 TO 255) =  ";SV
60  INPUT "ROTATION VALUE (0 TO 255) =  ";RV
70  HGR2 : HCOLOR= 3
80  SCALE= SV: ROT= RV
90  DRAW SD AT 140,90
100  FOR X = 1 TO 3000: NEXT X
110  HGR2 : TEXT : HOME
120  INPUT "WANT ANOTHER COMBINATION? (Y/N)  ";Y$
130  HOME
140  IF Y$ = "Y" THEN 40
150  END
```

Lines 10-30: The shape table is loaded and the starting address of the table is placed in the appropriate memory locations. CHR$(4) is assigned to the variable D$ in line 10. When this is done, D$ can be used to signal deferred DOS commands rather than CHR$(4). Note the difference between line 20 in Scale and Rotation and line 20 in Color trial.

Lines 40-60: These lines obtain the values for the shape, scale, and rotation that are to be used to create the display. Shapes appear larger when they are displayed with rotation values that are not multiples of 16 than when they are displayed with rotation values that are. The points along the vector path of a shape are further apart when the vectors are rotated at other than 90-degree angles. The reason is that the diagonal distance between points on the screen is greater than the vertical or horizontal distance between points.

Use the triangular shape (3) and the Nebraska shape (4) with several different scale and rotation values. The Nebraska shape was designed with only plotting vectors. When its size is increased by changes of scale value, its form is continuous. The triangular shape, however, has several vectors in its definition that do not plot. When scale values other than 1 are used to display the shape, gaps in its design are visible. This is because each of the vectors in the path of a shape are used SCALE number of times to draw the shape. This fact is even more obvious when a solid shape, such as the maple leaf (5), is displayed using scale values other than 1.

Lines 70-90: These lines set SCALE and ROT to the specified values and display the desired shape at the center of the screen.

Lines 100-150: These lines function similarly to lines 150-190 of the Color Trial program.

Simulating Motion

Motion can be simulated with shape tables by displaying a shape, leaving it on the screen for a brief moment, removing it, and then immediately displaying it again in a new location. The program in Listing 5-4 may be used to experiment with the concept of motion. The shape definition, spacing interval (number of dots between consecutive

Listing 5-4. Motion program.

```
5  REM     MOTION
10  HOME
20  PRINT  CHR$ (4)"BLOAD SAMPLE SHAPES"
30  POKE 232,0: POKE 233,48
40  INPUT "SHAPE DEFINITION (1 TO 5) =  ";S
50  INPUT "SPACING INTERVAL (1 TO 50) =  ";I
60  INPUT "EXPOSURE TIME (POSITIVE NUMBER) =  ";ET
70  HGR2 : HCOLOR= 3
80  ROT= 0: SCALE= 1
100 R = 90: REM   COORDINATE FOR DISPLAYING SHAPE
110  FOR C = 30 TO 250 STEP I
120  DRAW S AT C,R
130  FOR T = 1 TO ET: NEXT T: REM  EXPOSURE TIME
140  XDRAW S AT C,R: REM  ERASES SHAPE DRAWN IN LINE 1
     20
160  NEXT C
170  TEXT : HOME
180  INPUT "DO YOU WANT TO CONTINUE EXPERIMENTING?  (Y
     /N)  ";Y$
190  IF Y$ = "Y" THEN 40
200  END
```

drawings), and exposure time (time the image is left on the screen) are selected while running the program.

Some concepts you should consider in simulating motion with shapes are these:

1. Small shapes move more smoothly than larger shapes.
2. As the spacing increases, the rate of motion increases.
3. Shapes appear to flash when short exposure times are used.

A good set of values to begin experimenting with the Motion program is shape definition 1, spacing interval 2, and exposure time 8.

As was the case with the previous demonstration programs, the disk with Sample Shapes must be in the drive while Motion is running.

We will discuss lines 40 through 160 of the Motion program. If you do not understand the functions performed by the other lines of the program, you should refer to the discussion of the Color Trial program.

Line 40: The number of the shape that is to be used for the motion sequence is obtained and assigned to the variable S. Observe the motion sequence when the triangular shape (1) and the maple leaf shape (5) are used. The computer requires a significantly greater amount of time to follow all the instructions in the shape definition of the maple leaf than it does for the triangular shape. Therefore, motion performed using the maple leaf is not as smooth in appearance.

Line 50: The spacing interval determines the distance between consecutive displays of a shape. When a number is specified by the user, it is assigned to the variable I. If small spacing intervals are used, the time required to move the shape across the screen is greater than if large spacing intervals are used. However, large spacing intervals cause the motion to appear somewhat jerky.

Line 60: ET is assigned the value of the specified Exposure Time. The amount of time a shape is left on the screen before it is removed from the screen affects the speed of the motion. Small ET values cause the speed to increase. While decreasing the exposure time and decreasing the spacing interval should result in motion that is both fast and smooth, such is not always the case. Using short exposure times with large shapes will cause the display to flash. Flashing occurs whenever the exposure time is so brief that the computer starts to erase the shape immediately after it is drawn. Large shapes take so long to draw and erase that short exposure times do not leave the entire shape on the screen long enough to be seen by the user. It is normally appropriate to use longer exposure times with larger shapes and adjust the spacing increment to regulate the rate of motion. Jerky motion is generally easier on the user's eyes than flashy motion.

Lines 70-80: The statement display the high-resolution screen and initialize the color, rotation, and scale to the appropriate values.

Line 100: The motion sequence that causes a shape to be moved across the middle of the screen. The row coordinate used for displaying the shape is assigned to the variable R.

Line 110-130: This loop causes the variable C to take on the values of all the column coordinates used for displaying the shape. The motion sequence begins with the shape in column 30. The remaining column coordinates for displaying the shape are determined by the variable I, which is used for the step parameter in the FOR-NEXT loop. Line 120 places the shape at the appropriate point on the screen, and the loop in line 130 causes the shape to be left on the screen for the specified exposure time.

Line 140: The XDRAW command is used to remove the shape from the screen. This command works very well if the vector path of the displayed shape is properly designed. Some shapes will not be completely erased by the command. For example, when either shape 3 or 5 of the Sample Shapes shape table is specified in line 40, some of the points of the shapes will remain lighted after line 140 is executed.

This incomplete erasure happens whenever more than one plotting vector is used to light a point along the vector path. XDRAW erases a point in the shape by changing the color used to light the point

to its complement color. If the shape definition indicates that a plotting vector left the point, XDRAW changes the point to its complement color, which causes the point to be erased. If another plotting point appears later in the shape definition, XDRAW again complements the point's color—which causes it to reappear on the screen.

Proper coding of the vector path eliminates the problems associated with the XDRAW command. However, it is often difficult to code the vector path of complicated designs so that no more than one plotting vector leaves each point on the path. Therefore, we recommend that the DRAW command be used for erasing shapes. This can be done by setting HCOLOR to the value of the background color then executing the same DRAW command that was used to place the shape on the screen. Replace Line 140 of the Motion program with the following lines in order to use this technique:

```
115  HCOLOR=3
140  HCOLOR=0: DRAW S AT C,R.
```

Line 115 must be added to change HCOLOR back to white before displaying the shape in line 120.

MIXING TEXT AND GRAPHICS

Although the computer was not designed to allow text characters to be displayed on the high-resolution graphics screens, there are no restrictions on the kinds of designs that can be defined in shape tables. Therefore, through the use of shape tables, text characters can be displayed in the high-resolution graphics. Some topics discussed in this section that are related to using shape tables in this way include text character shape table, keyboard graphics, and graphics input and output displays.

Text Character Shape Table

Interaction between the user and the computer is limited when the graphics screens are in use; many programs therefore avoid extensive use of high-resolution graphics. Page 1 allows four lines of text to be displayed under the graphics screen and page 2 uses the entire monitor screen for graphics displays (see Fig. 5-2).

Through the use of shape tables, this deficiency can virtually be eliminated. Figure 5-3 illustrates the kinds of displays that are possible on the high-resolution graphics screens when shape tables are used to imitate text characters.

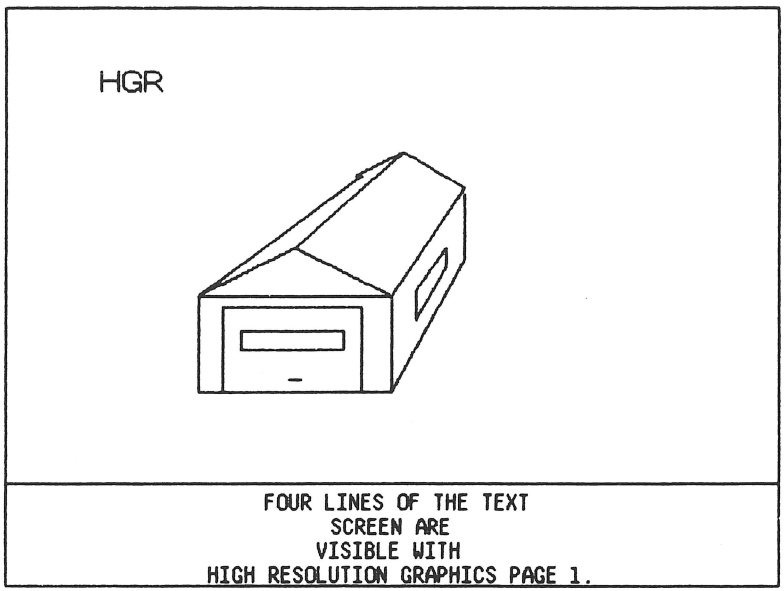
A shape table that includes the definitions for text characters is a useful programming aid, and one is provided in Listing 5-5. Enter the Character Set shape table starting at address $3000 and save it on disk by typing the following statement:

```
BSAVE CHARACTER SET, A$3000,
    L$2E0
```

The Character Set shape table contains shape definitions for most of the text characters. The characters and their corresponding positions in the shape table are presented in Table 5-2; while it is not a complete character set, it satisfies the needs of most programmers. Six bytes containing the hex code 00 appear at the end of the index. As many as three more character definitions can be appended to the shape table, and their relative addresses could be placed in the index where the 00 codes are located.

Programs that demonstrate some uses of the Character Set shape table are presented throughout the remainder of this chapter. The Draw 1 Character program (Listing 5-6) displays an exclamation point in the center of the screen. Different characters may be displayed by changing the shape number that immediately follows the DRAW commands in line 30. There are 60 shapes in the Character Set shape table, so any number from 1 to 60 is allowed after DRAW. The shape may be situated at different locations by changing the column and row coordinates in line 30 of the program.

An immediate reaction to this program may be that it is not very practical for displaying text on the high-resolution graphics pages in terms of programming time and effort. If it was necessary to specify the shape definition number and the column and row coordinates for each character displayed, there would be good reason to doubt the wisdom of using shape tables to mix text and graphics. The Character Display program (Listing 5-7) dem-



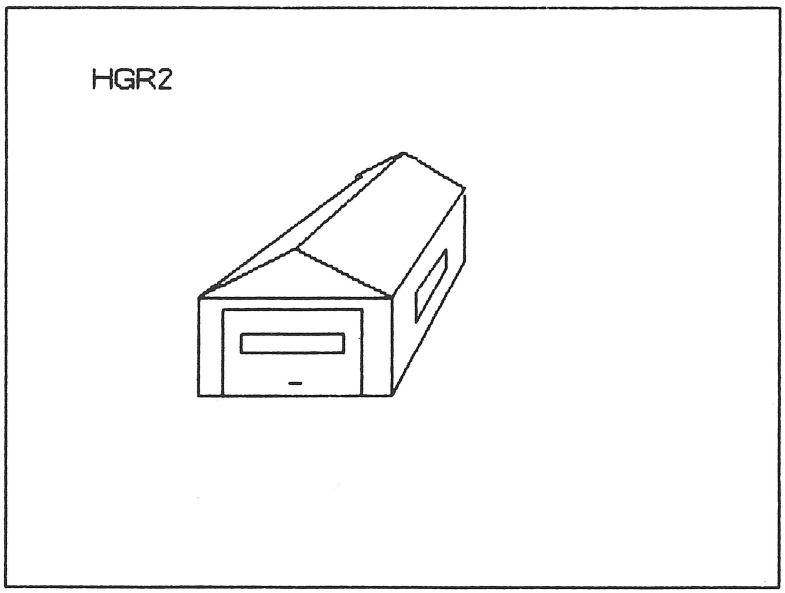


Fig. 5-2. This figure illustrates the types of displays that are available using the high-resolution graphics pages.

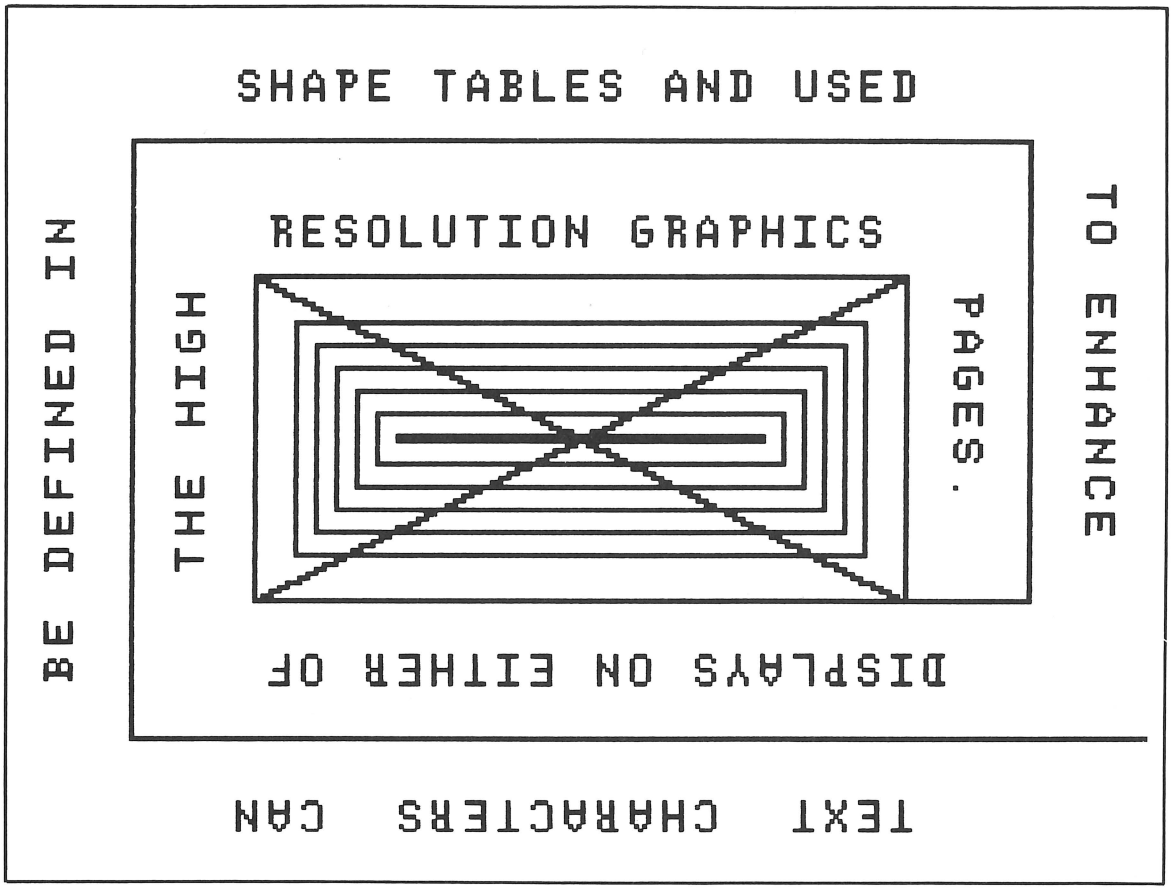


Fig. 5-3. Extraordinary displays can be achieved by using shape tables to represent the text characters for high-resolution graphics displays.

Listing 5-5. Hexadecimal listing of Character Set shape table (continues through page 86).

```
3000:3C 00 80 00 87 00 8D 00 9A 00 A7 00
B3 00 BE 00 C1 00 C9 00 D1 00 DD 00 E6
00 EA 00 EF 00 F2 00 FA 00 08 01 12 01
1C 01 27 01 30 01 3B 01 47 01 50 01 5C
01 65 01 69 01 6D 01 74 01 7D 01 84

:01 8D 01 9B 01 A8 01 B4 01 BF 01 CA 01
D6 01 E0 01 EB 01 F7 01 03 02 0E 02 1B
02 24 02 30 02 3C 02 49 02 52 02 5E 02
69 02 75 02 7E 02 89 02 95 02 A1 02 AD
02 B6 02 C2 02 C7 02 00 00 00 00 00 00

:93 0A 04 20 24 24 00 18 24 0D 36 06 00
2A AC 1E FE 24 67 3C 0C 6C BE 2D 05 00
```

```
DA 2A OE 2C 2C 1C 3F 1C OC OC 2E 2D 00
DC 20 35 4D 1E 8E 32 27 DF OC 04 00 63
E4 17 B6 76 6D 04 B8 07 20 00 20 24 00

:92 07 E0 24 OC OC 05 00 92 05 60 24 1C
1C 07 00 3C 1C 4C 6E 1E 16 15 D7 FC OC
25 00 24 9F 2A OE 2E 08 38 07 00 02 F6
07 00 4A 38 3F 3F 00 92 06 00 92 1B 64
OC OC OC 24 00 OC 04 38 BF 36 2E 1E OE

:2C OC 24 24 04 00 1B 60 OC 36 36 FE 2A
2D 25 00 65 E4 3F 17 56 BA 2E 2D 2D 00
DC 60 2D 15 F6 OE F6 3F 1C 04 00 3F 24
6C 31 36 F5 36 05 00 AD F6 3F 07 60 38
24 2C 2D 2D 00 AD F6 3F 07 20 05 38 24

:OC 2D 35 00 OC OC 3C 3F B7 8A 1E 3E 00
AD F6 3F 07 20 OC 1C 64 2D 15 36 00 E7
64 2D 15 BE 35 36 07 00 B8 11 07 00 16
1E 07 00 91 07 38 28 28 28 00 09 38 3F
BF OA 2D 2D 05 00 93 05 28 38 38 38 00

:CO 63 2D 15 F6 17 1C 06 00 43 36 2D 24
67 1C 3F 17 36 36 OE 2D 25 00 13 2D 56
24 24 1C 1C 17 17 36 36 07 00 37 BE 2D
2D 20 1C OC E4 3F 77 36 00 41 E0 3F 17
36 36 OE 2D OC 04 00 33 BE 2D 65 24 24
:1C 3F 77 36 00 25 08 38 3F 37 36 F5 36
2D 2D 04 00 25 08 38 3F 37 36 F5 36 05
00 2D 36 1E 3F 1C 24 24 OC 2D 06 00 2D
24 FC 1B 36 15 37 36 4D 21 24 00 24 OD
38 3F B7 52 31 9F 2D 2D 04 00 CO 03 28

:2D F5 36 36 1E 3F 20 00 60 05 F8 1B 36
15 37 36 4D E1 1C 04 00 CO 18 36 36 36
2D 2D 04 00 E4 1C 36 36 36 4D 21 24 24
BC 06 00 96 1B 24 24 E5 6C 09 36 BE 35
36 00 09 24 1C 3F 17 36 36 OE 2D 05 20

:04 00 61 E4 3F 77 AE 37 36 05 00 72 OC
24 E4 3F 17 36 36 OE 6D 04 00 37 36 4D
24 1C OC E4 3F 77 36 00 4C E0 3F 17 76
2D 15 F6 3F 07 20 00 24 OD 38 3F B7 52
31 36 00 09 24 FC 1B 36 36 76 2D OC 24

:00 09 24 FC 1B 36 36 OE OE OC OC 04 00
AE OE 24 24 24 DF 33 36 AE 37 07 00 OC
OC FC 1B AE 16 17 6E 09 E4 04 00 OC OC
FC 1B AE 56 36 07 00 C4 03 28 2D 35 17
```

```
D6 BB 2E 2D 25 00 D2 93 2D 2D 00 36 2E

:25 27 25 27 25 3C 3F 37 35 37 36 2E 24
24 25 2D 00
```

Table 5-2. Character Set Shapes.

SHAPE	POSITION	SHAPE	POSITION	SHAPE	POSITION
!	1	5	21	I	41
"	2	6	22	J	42
#	3	7	23	K	43
$	4	8	24	L	44
%	5	9	25	M	45
&	6	:	26	N	46
'	7	;	27	O	47
(	8	<	28	P	48
)	9	=	29	Q	49
*	1Ø	>	3Ø	R	5Ø
+	11	?	31	S	51
,	12	@	32	T	52
-	13	A	33	U	53
.	14	B	34	V	54
/	15	C	35	W	55
Ø	16	D	36	X	56
1	17	E	37	Y	57
2	18	F	38	Z	58
3	19	G	39	_	59
4	2Ø	H	4Ø	■	6Ø

Listing 5-6. Draw 1 Character program.

```
5  REM   DRAW 1 CHARACTER
10  PRINT  CHR$ (4);"BLOAD CHARACTER SET": SCALE= 1: ROT=
    0
20  POKE 232,0: POKE 233,48: HCOLOR= 3: HGR2
30  DRAW 1 AT 139,95: END
```

Listing 5-7. Character Display program.

```
5   REM    CHARACTER DISPLAY
10 D$ =  CHR$ (4)
20  PRINT D$"BLOAD CHARACTER SET"
30  HCOLOR= 3: SCALE= 1: ROT= 0
40  HGR2 : POKE 232,0: POKE 233,48
45 R = 20:C = 0
50  FOR S = 1 TO 60
55 C = C + 10
60  IF C > 270 THEN C = 10:R = R + 10
70  DRAW S AT C,R
80  NEXT S
90  END
```

onstrates how all the characters in the Character Set shape table can be displayed by placing the DRAW statement within a loop.

You should be very familiar by now with lines 10 through 40 of the Character Display program. Lines similar to these have been used in all the programs which have used shape tables. However, lines 45 through 80 require brief explanation.

Line 45: This line initializes the row and column coordinate variables (R and C). Since C is assigned a value of 0, it does not really need to be initialized. Applesoft BASIC initializes all variables to 0 whenever the RUN instruction is used.

Line 50: A loop to display all 60 shapes in the Character Set shape table is initialized in line 50. The variable S designates which of the 60 shapes is to be drawn with each pass through the loop.

Lines 55-60: Each of the shapes of the table is contained within a five-by-seven dot matrix. Line 55 causes the column coordinate, used for displaying the Sth shape of the table on the screen, to be increased by 10. As a result, the shapes will be separated horizontally by five screen dots. Line 60 checks to see if the Sth shape will fit on the current row. If the column coordinate exceeds 270, the row coordinate is increased by 10 and the column coordinate is reset to 10, causing the program to start a new row for displaying characters.

Lines 70-90: Line 70 displays the current shape at the appropriate point of the screen. After all 60 shapes have been displayed, the program ends. This technique for displaying shapes from the character set are practical only if the programmer knows which characters are to be placed on the screen. This is not always possible, since some of the characters may need to be selected by the user while the program is running. One method to facilitate user selection of the characters is demonstrated by the Keyboard Graphics program.

Keyboard Graphics

The Keyboard Graphics program (Listing 5-8) demonstrates how easily characters can be selected by the user and displayed on the graphics screen from within a program. After entering the program, place the disk with the Character Set shape table

Listing 5-8. Keyboard Graphics program listing.

```
5  REM   KEYBOARD GRAPHICS
6  REM
10  HOME :D$ =  CHR$ (4): POKE 232,0: POKE 233,48
20  PRINT D$;"BLOAD CHARACTER SET"
30 HL = 275:VL = 187:I = 8:SP = 3:CURS = 59:BLANK = 60
40  ROT= 0: SCALE= 1: HCOLOR= SP
50  HGR2 :ROW = SP:COLUMN = SP
60  DRAW CURS AT COLUMN,ROW: GET CH$: XDRAW CURS AT CO
     LUMN,ROW
70  IF CH$ = "" THEN 60
80 CH =  ASC (CH$)
90  IF CH > 32 AND CH < 91 THEN  GOSUB 240: DRAW CH -
     32 AT COLUMN,ROW: GOTO 170
100  IF CH = 13 THEN ROW = ROW + I:COLUMN = SP: GOTO 1
     90
110  IF CH = 3 THEN 250
120  IF CH = 21 THEN 170
130  IF CH = 32 THEN  GOSUB 240: GOTO 170
140  IF CH = 8 THEN  GOSUB 210
150  IF CH = 5 THEN 40
160  GOTO 60
170 COLUMN = COLUMN + I
180  IF COLUMN > HL THEN COLUMN = SP:ROW = ROW + I
190  IF ROW > VL THEN ROW = SP
200  GOTO 60
210  IF COLUMN > SP THEN COLUMN = COLUMN - I: RETURN
220  IF ROW > 3 THEN ROW = ROW - I:COLUMN = HL
230  RETURN
240  HCOLOR= 0: DRAW BLANK AT COLUMN,ROW: HCOLOR= SP: RETURN
250  END
```

into the disk drive, save the program, type RUN, and then start typing. As keys are pressed on the keyboard, the shape definition which corresponds to the appropriate character is displayed on the high-resolution graphics screen.

This program is intended for display purposes only; any programming commands entered will not be executed by the computer. For example, if HPLOT 50,50 is entered while running the Keyboard Graphics program, the Apple will not light the screen dot at coordinates 50,50.

Listed below are some characteristics and control features of the Keyboard Graphics program.

1. The cursor is an underline shape that does not blink.
2. Only 35 characters are displayed on each row. There are still 24 rows.
3. Instead of scrolling when the bottom line of the screen has been filled, the cursor just returns to the top line of the screen.
4. Repeat, Return, Right Arrow, and Left Arrow all retain their normal operations.
5. Press CTRL-C to exit the program.
6. Press CTRL-E to erase the screen (similar to HOME in Applesoft BASIC).

Now it's time for a detailed discussion of the

Keyboard Graphics program. It is important to understand how the program works, since many of the programming strategies are used in other programs presented later in the chapter.

Lines 10-20: These lines load the Character Set shape table and place its starting address in the appropriate memory locations. The table will be loaded into memory at $3000.

Line 30: Many of the key variables are initialized in line 30. HL is the variable that is used to set the right margin or Horizontal Limit of the screen. No characters will be placed to the right of column 275 of the high-resolution graphics screen. VL is the variable that is used to set the bottom margin of the screen. No characters will be placed below row 187 of the screen. The variable I is used to control both the horizontal and vertical Interval between displayed characters. The Interval used in this program is eight screen dots. Since all the text character definitions fit in a five-by-seven dot matrix characters displayed on any row of the screen will be separated by three dots, and one dot will separate the rows of characters.

SP is the variable which indicates the horizontal and vertical Starting Point for displaying shapes on the screen. Three is assigned to SP; a smaller value for SP would cause the first row of characters to extend above the top of the screen. Shape number 59 of the Character Set shape table, an underline character that serves as the cursor, is assigned to the variable CURS. Shape number 60 of the Character Set shape table, a six-by-nine dot solid block, is assigned to the variable BLANK. It is used as the space character in the program, and was defined on a larger matrix than the other characters so that it could be used to erase the cursor (59) as well as all the other characters.

Line 40: The rotation, scale, and color functions are set to their initial values in Line 40. White was the color chosen for displaying the characters. When other colors are used, some of the characters become unreadable due to the color properties of high-resolution graphics.

Line 50: Page 2 of high-resolution graphics is used for displaying the shapes because the shape table is stored in the memory segment reserved for page 1. ROW is the variable that keeps track of the row coordinate used for displaying a shape. It is initialized to 3, which is the coordinate of the starting point (SP) of the first text row displayed by the program. COLUMN is the variable which stores the column coordinate used for displaying a shape. Its initial value is also 3, since the first character of each row will be drawn at the third column of the high-resolution graphics screen.

Line 60: The first DRAW statement in line 60 causes the cursor to be displayed at the screen position used for displaying the next character typed at the keyboard. GET CH$ causes the program to wait until a key is pressed; when a key is pressed, its corresponding CHaracter is assigned to the variable CH$. The cursor shape (60) was designed so that no more than one plotting vector departed from each lighted point of the vector path. Therefore the XDRAW command can be used here to erase the cursor from the screen.

Lines 70-80: Line 70 checks to see if a null character (i.e., no character between the quotes) has been assigned to the variable CH$. If a null string has been entered, the keystroke is ignored and line 60 is executed again. The reason this check is included in the algorithm is that the null character has an ASCII code of 0. The BASIC statement (line 80) that translates an ASCII character to an ASCII code cannot evaluate the null character, and an ILLEGAL QUANTITY error occurs if CH$=0.

The null character is available from the keyboard by processing CTRL-@or

<ESC> <SHIFT> <P>

GET only obtains one keystroke from the keyboard, which should prevent the null character from being assigned to CH$. Sometimes the character can be generated accidentally, however, when two or more keys are pressed at the same time. The program would function properly without line 70 most of the time; it is included merely as a safeguard. The ASCII value of the character as-

signed to CH$ is assigned to the variable CH in line 80.

Line 90: Characters with ASCII codes between 32 and 91 are defined in the Character Set shape table. If the value of CH is in the range, the program displays the corresponding shape. Before a shape is displayed, shapes that may have been previously placed at that position must be erased. This is accomplished when line 90 calls the subroutine in line 240, which draws the space or blank character using the color black. The program then branches to line 170, where the coordinates for displaying the next character are determined. If the value of CH is not in the specified range, the program advances to line 100.

In order to display the proper shape, 32 must be subtracted from the ASCII code of the character to determine its position in the Character Set shape table. For example, the exclamation point (!) has the ASCII code 33, but it is shape number 1 in the shape table. Therefore the exclamation point's position can be determined by subtracting 32 from its ASCII code. In order for this transformation to work for all characters in the shape table, the ASCII codes of the characters defined in the shape table must be considered when their positions within the table are planned. (An ASCII code chart appears in Appendix A of this book.)

Line 100: This line checks to see if the Return key has been pressed. If it has, the ROW variable is increased by I and the COLUMN variable is reset to SP. The result is that the next character typed will appear in the first column of the next available row of the display screen. The program then branches to line 190, where it checks to see if there is enough room left to display a new row.

Line 110: When the user presses CTRL-C, the program ends. Line 110 checks to see if the CTRL-C character was pressed. If it was, a branch is taken to line 250, where the program ends.

Line 120: The Right Arrow key has an ASCII code of 21. If it is pressed, line 120 causes the program to branch to line 170, where the cursor is moved one position to the right on the screen.

Line 130: The space character cannot be displayed in line 90 because it must be drawn using the color black. Line 130 therefore causes the computer to execute the draw statement in line 240 before branching to line 170 to move the cursor to the next screen position.

Line 140: If CH has a value of 8, the Left Arrow key was pressed. The subroutine at line 210, executed when the Left Arrow key is pressed, instructs the computer to move the cursor back one screen position.

Line 150: Pressing CTRL-E causes the program to clear the high-resolution graphics screen and place the cursor in the upper left corner of the screen. Line 150 performs the required branch to line 40 when CH has a value of 5.

Line 160: If a key has been pressed that does not correspond to either a shape in the Character Set shape table or a recognized program instruction, the program ignores the keystroke by returning to line 60 without advancing the cursor to the next screen position.

Lines 170-200: Line 170 is executed after each key press that causes a shape to be displayed on the screen. It increases the column coordinate for displaying a shape by the value stored in the variable I. Line 180 checks to see if the column coordinate is outside the right margin (HL). If it is, instructions are executed that cause a new row to be started. Line 190 causes the cursor to be placed in the first display row of the screen if the value of ROW exceeds the vertical limit (VL). The computer unconditionally branches to line 60 when line 200 is executed.

Lines 210-230: The subroutine contained in these lines is executed whenever the Left Arrow key is pressed. Line 210 causes the cursor to retreat one screen position if the move does not cause it to move outside the left edge of the screen. If COLUMN is decreased in line 210, the subroutine is terminated. Line 220 causes the cursor position to be moved up one display row if the move does not cause it to be placed above the top of the screen. Row 230 exits the subroutine. The cursor position does not change if it is in the upper left corner of the screen.

Line 240: This subroutine is executed prior to

placing any character on the screen. It draws the space character (60) at the current cursor position using the color black. The result is that any character that may have previously occupied that position is erased. The subroutine is also executed when the space bar is pressed.

Line 250: The END statement is not optional in the Keyboard Graphics program, since the logical end and the physical end of the program do not coincide.

The Keyboard Graphics program can be altered so that the characters are displayed vertically, diagonally, or even upside down. The Vertical Keyboard program (Listing 5-9) operates like the Keyboard Graphics program, but the characters displayed are rotated 90 degrees counterclockwise and the cursor's home position is the lower left corner of the screen.

The following modifications were made to the Keyboard Graphics program to obtain the Vertical Keyboard program:

1. In line 40, ROT=48 replaced ROT=0.
2. ROW=VL replaced ROW=SP in line 50.
3. Line 100 was changed to read
   IF CH=13 THEN COLUMN=COLUMN+1: ROW=VL: GOTO 190
4. Line 170 needs to be ROW=ROW–I.
5. Line 180 should read
   IF ROW=SP THEN ROW<VL: COLUMN=COLUMN+I
6. In line 190, replace the variable ROW with

Listing 5-9. Vertical Keyboard.

```
5  REM ** VERTICAL KEYBOARD **
10  HOME :D$ =  CHR$ (4): POKE 232,0: POKE 233,48
20  PRINT D$;"BLOAD CHARACTER SET"
30 HL = 275:VL = 187:I = 8:SP = 3:CURS = 59:BLANK = 60
40  ROT= 48: SCALE= 1: HCOLOR= SP
50  HGR2 :ROW = VL:COLUMN = SP
60  DRAW CURS AT COLUMN,ROW: GET CH$: XDRAW CURS AT CO
    LUMN,ROW
70  IF CH$ = "" THEN 60
80 CH =  ASC (CH$)
90  IF CH > 32 AND CH < 91 THEN  GOSUB 240: DRAW CH -
    32 AT COLUMN,ROW: GOTO 170
100  IF CH = 13 THEN COL = COL + I:ROW = VL: GOTO 190
110  IF CH = 3 THEN 250
120  IF CH = 21 THEN 170
130  IF CH = 32 THEN  GOSUB 240: GOTO 170
140  IF CH = 8 THEN  GOSUB 210
150  IF CH = 5 THEN 40
160  GOTO 60
170 ROW = ROW - I
180  IF ROW < SP THEN ROW = VL:COL = COL + I
190  IF COL > HL THEN COL = SP
200  GOTO 60
210  IF ROW < VL THEN ROW = ROW + I: RETURN
220  IF COL > SP THEN COL = COL - I:ROW = SP
230  RETURN
240  HCOLOR= 0: DRAW BLANK AT COLUMN,ROW: HCOLOR= SP: RETURN
250  END
```

the variable COLUMN.

7. Line 210 now reads
   IF ROW<VL THEN ROW=ROW+I: RETURN
8. Line 220 interchanges the variables ROW and COLUMN and substitutes SP for HL.

Other modifications can be made to Keyboard Graphics for displaying the characters using values of SCALE greater than 1. The set of characters of a foreign language also could be defined in a shape table and displayed.

Graphics Input and Output Displays

In the Keyboard Graphics program only text characters are displayed on the screen. Similar techniques can be used to mix graphics and text, thus increasing programming options. Though the program developed for illustrating these techniques has an educational theme, they are appropriate for all types of programs. Innovative programs can be created by using Keyboard Graphics program features to:

1. Display graphics and text on the same screen.
2. Permit user responses to be displayed on the graphics screen.
3. Evaluate user input and display responses on the graphics screen.

The Mixer program (Listing 5-10) demonstrates how to write program utilize both text and graphics producing the display shown in Fig. 5-4. In addition, a user's reply and an appropriate computer response will be displayed. Graphics displays should always be planned *before* writing the code.

Listing 5-10. Mixer.

```
5  REM **   MIXER PROGRAM   **
10  HOME :D$ =  CHR$ (4): POKE 232,0: POKE 233,48
20  PRINT D$;"BLOAD CHARACTER SET"
30 HL = 275:VL = 187:I = 8:SP = 3: HCOLOR= 3: ROT= 0: SCALE=
 1
40  DIM SR$(35):CURS = 59::BLANK = 60: HGR2
50  HPLOT 32,24 TO 80,24 TO 80,72 TO 32,72 TO 32,24
60 CH$ = "THIS POLYGON HAS"
70 DC = 15:DR = 4: GOSUB 200
80 CH$ = "FOUR CONGRUENT SIDES"
90 DC = 14:DR = 6: GOSUB 200
100 CH$ = "AND"
110 DC = 22:DR = 8: GOSUB 200
120 CH$ = "FOUR CONGRUENT ANGLES."
130 DC = 14:DR = 10: GOSUB 200
140 CH$ = "1.  NAME THE POLYGON > "
150 DC = 3:DR = 13: GOSUB 200
160 DC = 26::DR = 13: GOSUB 300
170 CH$ = "RIGHT": IF REPLY$ <  > "SQUARE" THEN CH$ =
     "WRONG"
180 DC = 3:DR = 15
190  GOSUB 200: END
200 ROW = SP + (DR - 1) * I:COLUMN = SP + (DC - 1) * I
210  FOR X = 1 TO  LEN (CH$)
220  IF  MID$ (CH$,X,1) =  CHR$ (32) THEN 240
```

```
230  DRAW  ASC ( MID$ (CH$,X,1)) - 32 AT COLUMN,ROW
240 COLUMN = COLUMN + I
250  NEXT X: RETURN
300 COLUMN = SP + (DC - 1) * I:ROW = SP + (DR - 1) * I
    :RC = 1:REPLY$ = ""
310  DRAW CURS AT COLUMN,ROW: GET CH$: XDRAW CURS AT C
    OLUMN,ROW
320  IF CH$ = "" THEN 310
330 CH =  ASC (CH$)
340  IF CH > 32 AND CH < 91 THEN  GOSUB 500: DRAW CH -
    32 AT COLUMN,ROW: GOTO 400
350  IF CH = 13 THEN 510
360  IF CH = 21 THEN 400
370  IF CH = 32 THEN  GOSUB 500: GOTO 400
380  IF CH = 8 THEN  GOSUB 440
390  GOTO 310
400 RC = RC + 1:COLUMN = COLUMN + I
410  IF COLUMN > HL THEN COLUMN = SP:ROW = ROW + I
420  IF ROW > VL THEN RC = RC - 1: GOTO 510
430  GOTO 310
440  IF ROW > SP + (DR - 1) * I THEN 470
450  IF COLUMN > SP + (DC - 1) * I THEN RC = RC - 1:CO
    LUMN = COLUMN - I
460  GOTO 490
470  IF COLUMN > SP THEN COLUMN = COLUMN - I:RC = RC -
    1: GOTO 490
480 ROW = ROW - I:COLUMN = HL:RC = RC - 1
490  RETURN
500  HCOLOR= 0: DRAW BLANK AT COLUMN,ROW: HCOLOR= SP:S
    R$(RC) = CH$: RETURN
510  FOR X = COLUMN TO HL STEP I: HCOLOR= 0: DRAW BLAN
    K AT X,ROW: NEXT X: HCOLOR= SP
515  IF RC > 255 THEN RC = 255
520  FOR X = 1 TO RC - 1
530 REPLY$ = REPLY$ + SR$(X)
540  NEXT X
550  RETURN
```

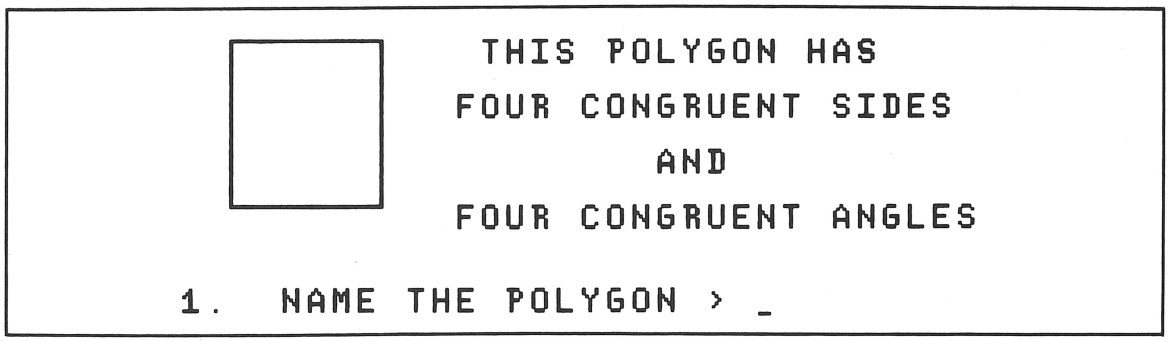


Fig. 5-4. This display is created by the Mixer program.

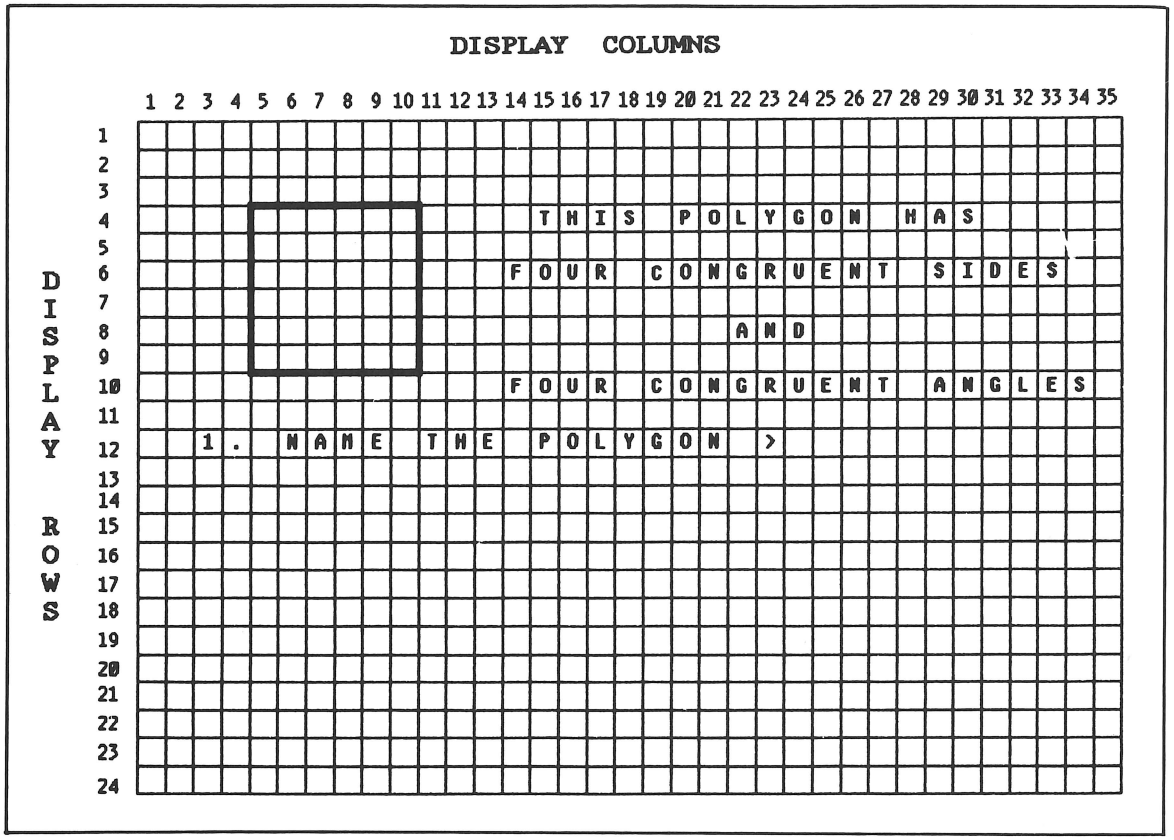


Fig. 5-5. High-Resolution graphics displays should be sketched on graph paper before the program is written.

Lines 10-40: These lines load the Character Set shape table and initialize the variables used throughout the program.

Line 50: This line creates the square. You should use graph paper to plan your display (see Fig. 5-5).

Lines 60-150: These lines of the Mixer program assign each line of text to a string variable (CH$) and declare the display column (DC) and display row (DR) where the first character of the string is to be placed. After each string assignment and its display coordinates have been declared, the statement GOSUB 200 is executed.

Lines 200-250: This subroutine, called by the previous section of code, converts the display coordinates assigned to the variables DR and DC to high-resolution graphics coordinates, and then draws each shape needed to produce the text.

Lines 300-550: This long subroutine, called by line 160 and featuring a number of possible RETURNs to mainline code, is provided to display the user's response. If you compare it to the Keyboard Graphics program, you will find them very similar.

Line 160: Before the above subroutine is executed, the display coordinates for the reply are declared in line 160. The user reply is restricted to 35 characters; longer responses can be accommodated by changing the DIM statement in line 10. The subroutine retains the reply in the SR$ array, then reassigns the array contents to the string variable REPLY$.

Lines 170-190: These lines evaluate the reply, display the appropriate response, and end the program.

The Mixer program can be restructured for more efficient execution. For example, the version

of Mixer in Listing 5-11 has major subroutines for displaying the text and user input near the beginning of the program.

Other programming strategies can be used to assign the text and display coordinates to the appropriate variable. The information could be placed

Listing 5-11. Alternate Mixer.

```
5  REM  **   ALTERNATE MIXER   **
10  GOTO 600
200 ROW = SP + (DR - 1) * I:COLUMN = SP + (DC - 1) * I
210  FOR X = 1 TO  LEN (CH$)
220  IF  MID$ (CH$,X,1) =  CHR$ (32) THEN 240
230  DRAW  ASC ( MID$ (CH$,X,1)) - 32 AT COLUMN,ROW
240 COLUMN = COLUMN + I
250  NEXT X: RETURN
300 COLUMN = SP + (DC - 1) * I:ROW = SP + (DR - 1) * I
    :RC = 1:REPLY$ = ""
310  DRAW CURS AT COLUMN,ROW: GET CH$: XDRAW CURS AT C
    OLUMN,ROW
320  IF CH$ = "" THEN 310
330 CH =  ASC (CH$)
340  IF CH > 32 AND CH < 91 THEN  GOSUB 500: DRAW CH -
    32 AT COLUMN,ROW: GOTO 400
350  IF CH = 13 THEN 510
360  IF CH = 21 THEN 400
370  IF CH = 32 THEN  GOSUB 500: GOTO 400
380  IF CH = 8 THEN  GOSUB 440
390  GOTO 310
400 RC = RC + 1:COLUMN = COLUMN + I
410  IF COLUMN > HL THEN COLUMN = SP:ROW = ROW + I
420  IF ROW > VL THEN RC = RC - 1: GOTO 510
430  GOTO 310
440  IF ROW > SP + (DR - 1) * I THEN 470
450  IF COLUMN > SP + (DC - 1) * I THEN RC = RC - 1:CO
    LUMN = COLUMN - I
460  GOTO 490
470  IF COLUMN > SP THEN COLUMN = COLUMN - I:RC = RC -
    1: GOTO 490
480 ROW = ROW - I:COLUMN = HL:RC = RC - 1
490  RETURN
500  HCOLOR= 0: DRAW BLANK AT COLUMN,ROW: HCOLOR= SP:S
    R$(RC) = CH$: RETURN
510  FOR X = COLUMN TO HL STEP I: HCOLOR= 0: DRAW BLAN
    K AT X,ROW: NEXT X: HCOLOR= SP
520  FOR X = 1 TO RC - 1
530 REPLY$ = REPLY$ + SR$(X)
540  NEXT X
550  RETURN
600  HOME :D$ =  CHR$ (4): POKE 232,0: POKE 233,48
610  PRINT D$;"BLOAD CHARACTER SET"
```

```
620 HL = 275:VL = 187:I = 8:SP = 3: HCOLOR= 3: ROT= 0:
    SCALE= 1
630  DIM SR$(35):CURS = 59::BLANK = 60: HGR2
640  HPLOT 32,24 TO 80,24 TO 80,72 TO 32,72 TO 32,24
650 CH$ = "THIS POLYGON HAS"
660 DC = 15:DR = 4: GOSUB 200
670 CH$ = "FOUR CONGRUENT SIDES"
680 DC = 14:DR = 6: GOSUB 200
690 CH$ = "AND"
700 DC = 22:DR = 8: GOSUB 200
710 CH$ = "FOUR CONGRUENT ANGLES."
720 DC = 14:DR = 10: GOSUB 200
730 CH$ = "1.  NAME THE POLYGON > "
740 DC = 3:DR = 13: GOSUB 200
750 DC = 26::DR = 13: GOSUB 300
760 CH$ = "RIGHT": IF REPLY$ <  > "SQUARE" THEN CH$ =
    "WRONG"
770 DC = 3:DR = 15
780  GOSUB 200: END
```

Listing 5-12. Enhancement module for Alternate Mixer.

```
650  FOR Y = 0 TO 4
660  READ CH$,DC,DR: GOSUB 200
680  NEXT Y
750 DC = 26::DR = 13: GOSUB 300
760 CH$ = "RIGHT": IF REPLY$ <  > "SQUARE" THEN CH$ =
    "WRONG"
770 DC = 3:DR = 15
780  GOSUB 200: END
800  DATA  "THIS POLYGON HAS",15,4,"FOUR CONGRUENT SID
    ES",14,6,"AND",22,8,"FOUR CONGRUENT ANGLES.",14,1
    0,"1.  NAME THE POLYGON > ",3,13
```

in DATA statements or in sequential text files. For example, to use DATA statements replace lines 650 through 780 of the Alterate Mixer program with the lines shown in Listing 5-12.

ARCADE-STYLE PROGRAMS

Arcade-style programs can generally be classified as games, even though they may deliver some educational material. Good arcade games are designed so that the player has extensive control over the graphics displays. In order to achieve the desired interaction, the program must check the keyboard (or paddles) periodically for player input. The program must react immediately to any player response, and alter the graphics displays accordingly. Let's look at a simple program that demonstrates some of the elementary programming strategies used in arcade games.

The Arcade Space Game uses four graphics figures or shapes. The shapes represent the player's weapon, an enemy weapon, and appropriate discharges of projectiles for each weapon (see Fig. 5-6). The player's weapon is a mobile antispacecraft position that fires a small projectile.

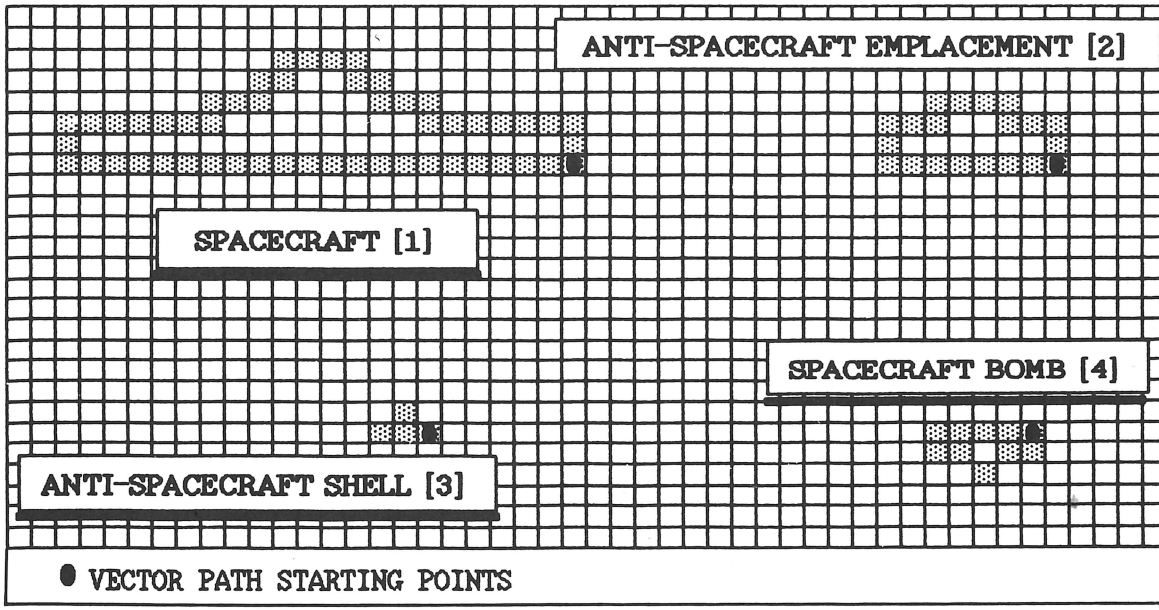


Fig. 5-6. These game pieces are stored in the Game Shapes shape table, which is used for the Arcade Game program.

The enemy weapon is an armed spacecraft that drops a small bomb. The shapes are stored in the shape table in this order: spacecraft, player's weapon, antispacecraft bullet, spacecraft bomb. The shape table containing the shapes is given in Listing 5-13, and should be entered starting at memory address $8000. The shape table should be stored on the disk using this command:

BSAVE GAME SHAPES, A$8000, L$3C

The Arcade Space Game allows the player to move the mobile antispacecraft weapons position by pressing the right and left arrow keys. An antispacecraft projectile can be fired by pressing the space bar. The enemy spacecraft will proceed from the upper left corner of the screen to the upper right corner of the screen, dropping bombs as it moves. If the projectiles collide, they disappear and the player is awarded 50 points. If the player hits the spacecraft, 25 points are awarded, the spacecraft disappears, and another spacecraft is positioned in the upper left corner of the screen. Should an enemy craft make it to the upper right corner of the screen, 50 points are deducted from the score, and the spacecraft makes another pass. Each time the player fires, 2 points are deducted from the score. Five points are deducted each time a bomb makes it to the bottom of the screen. If a bomb hits the player's weapon, the game ends. In order to simplify the demonstration program, only one bomb

Listing 5-13. Hexadecimal listing of shape table for Arcade Space Game.

```
8000:04 00 0A 00 25 00 30 00 34 00 3F 3F
3F 3F 3F 3F 3F 3F 3F 3F 27 2C 2D 2D 25
2D 2C 2C 2D 2E 2E 35 2D 2D 2D 36 00 3F
3F 3F 27 2C 25 2D 35 2D 36 00 3F 0C 05
00 3F 3F 2E 0E 05 28 05 00
```

and one antispacecraft projectile can be active at any time.

Several new programming strategies and statements are used in Listing 5-14, the Arcade Space Game; a line-by-line discussion helps explain the program listing. We suggest you enter and play the game before reading the line-by-line comments.

The technique of using HIMEM to protect selected areas of memory has already been explained. Although the program size and number of variables used in the Arcade Space Game does not require the use of the technique, it is utilized here for demonstration purposes.

Lines 30-60: HIMEN is set at 32767. This

Listing 5-14. Arcade Space Game.

```
10  REM     ARCADE SPACE GAME
20  REM     =================
30  HIMEM: 32767
40 D$ =  CHR$ (4): HOME
50  PRINT D$"BLOAD GAME SHAPES,A$8000"
60  POKE 232,0: POKE 233,128
70  REM
80  REM  SET UP INITIAL SCREEN
90  REM  ---------------------
100   HGR
110   HCOLOR= 3: SCALE= 1
120 RE = 10:CE = 24
130 RG = 155:CG = 143
140   DRAW 1 AT CE,RE
150   DRAW 2 AT CG,RG
160   HOME : VTAB 22: PRINT "PRESS <RETURN> TO START GA
      ME.";
170   GET G$: IF G$ <  >  CHR$ (13) THEN 170
180   HOME : VTAB 22: PRINT "SCORE:"
190   REM
200   REM    START GAME SEQUENCE
210   REM    -------------------
220   GOSUB 350: REM  MOVE ENEMY
230   IF  PEEK ( - 16384) = 160 THEN  GOSUB 390: REM  F
      IRE
240   IF  PEEK ( - 16384) = 136 THEN  GOSUB 640: REM  M
      OVE LEFT
250   IF  PEEK ( - 16384) = 149 THEN  GOSUB 660: REM
      MOVE RIGHT
260   GOSUB 520: REM  MOVE BOMB
270   GOSUB 490: REM  CHECK FOR COLLISION
280   GOSUB 440: REM  MOVE BULLET
290   GOSUB 490: REM  CHECK FOR COLLISION
300   VTAB 22: HTAB 8: PRINT SC"    ": REM  DISPLAY SCO
      RE
310   GOTO 220
320   REM
330   REM    GAME SUBROUTINES
340   REM    ----------------
```

```
350  HCOLOR= 0: DRAW 1 AT CE,RE:CE = CE + 2: IF CE > 2
     79 THEN CE = 24:SC = SC - 50
360  HCOLOR= 3: DRAW 1 AT CE,RE
370  IF  NOT BOMB THEN BOMB = 1:DR = 10:DC = CE: DRAW
     4 AT DC,DR
380  RETURN
390  IF FIRE THEN  RETURN
400 SC = SC - 2
410 FIRE = 1: POKE  - 16368,0
420 FR = 150:FC = CG
430  DRAW 3 AT FC,FR: RETURN
440  IF  NOT FIRE THEN  RETURN
450  HCOLOR= 0: DRAW 3 AT FC,FR:FR = FR - 10
460  IF FR < 0 THEN FIRE = 0: RETURN
470  HCOLOR= 3: DRAW 3 AT FC,FR
480  RETURN
490  IF  PEEK (234) THEN  GOSUB 570: REM  LOCATE COLLI
     SION
500  POKE 234,0
510  RETURN

520  IF  NOT BOMB THEN  RETURN
530  HCOLOR= 0: DRAW 4 AT DC,DR
540 DR = DR + 10: HCOLOR= 3: IF DR > 155 THEN BOMB = 0
     :SC = SC - 5: RETURN
550  DRAW 4 AT DC,DR
560  RETURN
570  HCOLOR= 0
580  IF  PEEK (226) < 15 THEN  DRAW 1 AT CE,RE: DRAW 3
      AT FC,FR:CE = 24:FIRE = 0: HCOLOR= 3:SC = SC + 2
     5: RETURN
590  IF  PEEK (226) < 141 THEN SC = SC + 50: DRAW 3 AT
     FC,FR: DRAW 4 AT DC,DR:FIRE = 0:BOMB = 0: RETURN
600  POP : VTAB 23: PRINT "GAME OVER!!!  -  PRESS <Q>
     OR <C> ";
610  GET G$: IF G$ <  > "C" AND G$ <  > "Q" THEN 610
620  IF G$ = "Q" THEN  HGR : TEXT : HOME : END
630 SC = 0:BOMB = 0:FIRE = 0: GOTO 70
640  IF CG < 10 THEN  RETURN
650  HCOLOR= 0: DRAW 2 AT CG,RG:CG = CG - 5: HCOLOR= 3
     : DRAW 2 AT CG,RG: RETURN
660  IF CG > 275 THEN  RETURN
670  HCOLOR= 0: DRAW 2 AT CG,RG:CG = CG + 5: HCOLOR= 3
     : DRAW 2 AT CG,RG: RETURN
```

means the BASIC program will not use the memory segment from $8000 to $95FF to store string values. Information placed in this segment by the programmer is therefore protected. The shape table is placed at $8000 in line 50. Line 60 places the location of the shape table in memory locations 232 and 233.

Lines 100-110: These lines cause page 1 of

high-resolution graphics to be displayed and set the color and scale values for drawing the shapes on the screen.

Lines 120-130: The variables RE and CE are used for the row and column coordinates, respectively, to draw the enemy spacecraft. The enemy shape will start the game in column 24. The row coordinate of the enemy shape, 10, is constant throughout the game. RG and CG are the variables used for the row and column coordinates of the mobile ground position. The row coordinate, 155, of the player's weapon also remains constant.

Lines 140-150: The ground weapon and target craft are displayed in their starting locations.

Lines 160-180: Line 160 instructs the player to press the Return key to begin the game. Line 170 stops the program until the return key is pressed. Line 180 clears the text screen and prints the word SCORE below the graphics screen.

Lines 220-310: These lines are responsible for controlling the game sequence. Subroutines that actually place the shapes on the screen and determine the scoring are called by these lines. The game would execute more quickly if the subroutine statements were integrated with the game sequence statements, but it is important for demonstration purposes that each step in the sequence be easy to isolate. Each line in the game sequence module of the program needs to be discussed separately.

The first step in the game sequence is to move the spacecraft by calling the subroutine that starts in line 350. Line 220 makes this call.

Keyboard presses are checked by the command PEEK (– 16384) in Line 230. If the space bar was the last key pressed, the value 160 will be contained in the memory location. This value would indicate that the player wants to fire a shot at the enemy target, which is accomplished in subroutine 390.

In line 240, a value of 136 in memory location – 16384 indicates that the left arrow key has been pressed, and that the player's weapon is to be moved to the left by subroutine 640. Line 250 calls subroutine 660, which moves the player's weapon to the right, whenever the right arrow key has been pressed.

Line 260 calls the subroutine that moves the spacecraft's bombs down on the screen. Each time the bomb is moved, the program checks in line 270 to see if the bomb hit the player's weapon. The subroutine in line 490 checks for any shape collisions.

Next, in line 280, the player's bullet must be moved towards the top of the screen by calling subroutine 440. After the player's bullet is advanced, the program must again check (line 290) for shape collisions to determine if either the spacecraft or a bomb has been struck.

The current score is displayed by line 300, and line 310 simply directs the computer to go through the game sequence again.

Lines 350-670: The subroutines in this module of the program cause shapes to be placed on the screen in appropriate locations, tabulate the scoring, and determine when the game is over. Each subroutine is discussed separately.

Lines 350-380: This subroutine removes the spaceship from the screen and calculates its new column coordinate. It is moved two columns to the right on the screen if enough room remains. If the move would cause it to move off the screen, it is placed back to its starting position, and 50 points are deducted from the score variable, SC. The subroutine also checks to see if a bomb has been dropped. The flag variable, BOMB, has a value of 1 if a bomb is currently on its way to the surface and a value of 0 if no bomb is in motion. If no bomb is active, the subroutine sets the drop row, DR, and drop column, DC, variables to the value for the first position on the dropped bomb's path. If the dropping action must be initiated, the BOMB flag is set to 1 and the bomb is drawn on the screen. The program then returns to line 230.

Lines 390-430: The flag variable FIRE is set to 1 whenever a bullet is fired. Since only one bullet can be in motion at any time, this subroutine is not executed if a bullet is already in motion. The check for an active bullet is made in line 390. Line 400 deducts two points from the score since the player is using a bullet. The FIRE flag is set to a value of 1 in line 410, and the keyboard strobe is cleared. The initial row coordinate FR and column coordinate FC are set in line 420. The bullet is drawn in its first

position of the firing path in line 430 before the subroutine is terminated.

Lines 440-480: When the FIRE flag has a value of 0, the subroutine in lines 440-480 is not executed. This subroutine moves the bullet up 10 screen rows. If the move would cause the bullet to be positioned above the enemy target, then the FIRE flag is cleared to 0 and the subroutine is exited in line 460. The illusion of motion is created by erasing the bullet in line 450, recalculating the row coordinate in line 450, then drawing it on the screen in its new location in line 470.

Lines 490-510: Memory location 234 can be checked to see if the last shape drawn overlaps or collides with other shapes on the screen. The value in 234 indicates the number of screen pixels that were involved in the collision. A value of 5 in location 234 means that 5 pixels that had been lighted were reused in displaying the most recently plotted shape. If no collision has occurred, the value of 234 is 0. If a collision has occurred, the program calls subroutine 570 to determine the screen positions of the pixels involved in the collision.

Lines 520-560: This subroutine controls the descent of the bombs dropped by the enemy spacecraft. If no bomb is active, the subroutine is ignored. If the new position of a bomb would be below the player's weapon, the bomb is removed from the screen, and 5 points are deducted from the score. The bomb is moved down the screen 10 rows at a time.

Lines 570-630: Memory location 226 always contains the row coordinate of the last plotted high-resolution pixel; locations 224 and 225 contain the column coordinate of the pixel. The subroutine in lines 570-630 determines which shapes were involved in the last collision. Only the row coordinate of the last shape drawn is needed to make the determination; thus only location 226 needs to be checked.

If the collision occurs above row 15 of the screen, then the spacecraft was involved. The only shape that can collide with the spacecraft is a bullet. Should this occur, line 580 sets the enemy targets column coordinate to its starting position, adds 25 points to the score, clears the FIRE flag to 0, and terminates the subroutine.

Line 590 performs the check necessary to determine if a bomb and a bullet have collided. If they have, then both the FIRE and BOMB flags are cleared, the corresponding shapes are removed from the screen, and 50 points are added to the score before the subroutine is terminated. If the program reaches line 600, then a bomb must have struck the player's weapon, ending the game. The player is prompted to press C to play another game or Q to quit. Lines 610 through 630 perform the necessary branching as determined by the player's response to the prompt.

Lines 640-670: The player's weapon is moved to the left 5 columns if enough room is available. If it cannot be moved, it is left in its current location. The subroutine in lines 660-670 moves the player's weapon to the right 5 screen columns if possible.

Whenever the collision counter, 234, is used in a program, extreme care must be taken in planning the paths that the shapes are to follow. This is another case where "close" does not count. At least one pixel of each shape must overlap at the point of impact in order for a collision to be detected. Notice that both the bullets and bombs in the game started on rows with coordinates that are multiples of 10. Both of the shapes are also moved in increments of 10 rows. Therefore, if both shapes are on columns that would allow them to hit each other, a collision will occur. A similar consideration must also be given to the positioning of the target and weapon.

As the Arcade Space Game is running, the shapes appear to be flashing as they are moved across the screen. If the game were modified so that more than one shot could be fired or a greater number of enemy craft were displayed, the flashing would become more obvious. Page flipping can be used to help eliminate flashing. The technique requires that no shapes be erased from the exposed graphics page. This means that in order to use page flipping, the computer must allow shapes to be drawn on the page that is not displayed. The statements needed to use page flipping with vector graphics are used in the Draw-N-Flip program, Listing 5-15.

Listing 5-15. Draw-N-Flip program listing.

```
10  REM      DRAW-N-FLIP
20  HIMEM: 32767
30 ST$ = "CHARACTER SET": HOME
40  PRINT  CHR$ (4)"BLOAD"ST$",A$8000"
50  POKE 232,0: POKE 233,128
60  SCALE= 1
70 NS = 55
80 S = 1
90 T = 1
100 DB = 5
110  REM
120  REM     WITH PAGE FLIPPING
130  REM     ------------------
140  HGR : HGR2 : POKE  - 16302,0
150 PG = 1
160  FOR R = 10 TO 180 STEP S
170  POKE 230,(PG) * 32
180  HCOLOR= 3: FOR C = 1 TO NS: DRAW C AT C * DB,R: NEXT

190  POKE  - 16301 + PG,0
200  POKE 230,(3 - PG) * 32
210  HCOLOR= 0: FOR C = 1 TO NS: DRAW C AT C * DB,R -
     S: NEXT
220 PG = 3 - PG
230  IF  PEEK ( - 16384) = 160 THEN 300
240  IF  PEEK ( - 16384) = 155 THEN  TEXT : END
250  NEXT R
260  GOTO 140
270  REM
280  REM    WITHOUT PAGE FLIPPING
290  REM    ---------------------
300  HGR : HGR2
310  POKE  - 16368,0
320  FOR R = 10 TO 180 STEP S
330  HCOLOR= 3: FOR C = 1 TO NS: DRAW C AT C * DB,R: NEXT

340  FOR Z = 1 TO T: NEXT Z
350  HCOLOR= 0: FOR C = 1 TO NS: DRAW C AT C * DB,R: NEXT

360  IF  PEEK ( - 16384) = 160 THEN  POKE  - 16368,0: GOTO
     140
370  IF  PEEK ( - 16384) = 155 THEN  TEXT : END
380  NEXT R: GOTO 320
```

The program Draw-N-Flip demonstrates the technique of page flipping when using the two high-resolution graphics screens. It also allows the user to compare the difference between motion scenes with and without page flipping. Any shape table could be used with the program, but the Character Set shape table is used so that more shapes are available for creating the display.

Enter and run the program. The program starts by displaying 55 shapes on row 10 of the screen. They are moved down, one row at a time, using page flipping, until they reach row 180. The sequence is then repeated. Pressing the space bar causes the same sequence to be executed without the benefit of page flipping. Pressing the space bar again causes the program to revert to using the page flipping technique. Pressing the Escape key at any time causes the program to end.

This program can be modified easily so that the user can observe the effects of changing several of the variables. NS is the variable used to store the number of shapes that will be displayed in the program. Changing the value of the variable in line70 to a smaller number reduces the flashing effect. The speed at which the row of shapes descends the screen can be altered by changing the values of either the variable S, the step parameter, or T, the time delay parameter. The distance between the shapes is controlled by the variable DB. If fewer shapes are displayed, DB can be increased to spread the shapes out along the row.

Lines 140 through 260 contain the statements necessary to accomplish page flipping. Let's look at each line separately.

Line 140: Both graphics pages are cleared before beginning the sequence. The statement POKE −16302,0 instructs the computer to display all $2000 bytes of the graphics pages, preventing text from being displayed at the bottom of the screen. This is necessary to keep part of the page 2 text memory from being displayed. Omitting the statement from the program is a good way to see why it is needed.

Line 150: PG keeps track of which page is to be used for displaying the shapes. It is assigned a value of 1 in line 150 so that the row of shapes will be drawn on page 1 first.

Line 160: This line initializes the loop that changes the row coordinate that will be used for drawing the shapes. The row variable R takes on the values 10 through 180. The step variable S has a value of 1, but it can be changed in line 80 to increase the shapes' rate of descent.

Line 170: Memory location 230 contains information which tells the computer which of the high-resolution pages is to be used for plotting points or drawing shapes. If the value in location 230 is 32, page 1 is used; if the value is 64, then page 2 is used. The formula PG*32 has a value of 32 when PG is 1, and a value of 64 when PG is 2. Line 170 causes the appropriate value for the desired page to be placed in memory location 230.

Line 180: After the color is set, the first 55 shapes in the Character Set shape table are drawn on the high-resolution page specified in line 170. It should be noted that the page upon which the shapes are drawn is not the page displayed; the shapes will not appear on the screen as they are drawn.

Line 190: The statement POKE −16299,0 causes page 2 to be displayed. POKE −16300 causes page 1 to be displayed. When PG has a value of 1, the expression −16301 + PG has a value of −16300. The expression's value is −16299 when PG is 2. The statement causes the computer to display the page used for drawing the shapes in line 180. As a result, the page that was previously displayed is hidden from view, but it is not erased.

Line 200: In order to erase the hidden high-resolution graphics screen without disturbing the displayed screen, the computer must be instructed to use the hidden page. The expression 3-PG has a value of 2 when PG is 1, and a value of 1 when PG is 2. The statement causes the appropriate value, 32 or 64, to be placed in memory location 230.

Line 210: After setting the color to black, the shapes on the hidden page are drawn with the same coordinates used to place them on the screen. This erases the hidden page. Notice that the row coordinate used for erasing the shapes is R-S.

Line 220: The value of the variable PG alternates between 1 and 2 each time line 220 is executed. Therefore the row of shapes is alternately drawn and displayed on pages 1 and 2 each time the loop is executed.

Lines 230-260: This line checks to see if the space bar has been pressed. If so, a branch is taken to the section of the program which does not use page flipping. If the Escape key has been pressed before line 240 is executed, the program ends. These two lines are not part of the page flipping

sequence. Line 250 causes the page flipping sequence to continue until row 180 of the screen is used for drawing the shapes, and line 260 causes the sequence to continue indefinitely.

Many entertaining programs can be developed using shape tables and the techniques discussed in this chapter. After sufficient practice and experimentation, most programmers should be able to meet their programming needs using these graphics techniques. These programs represent just a few of the things that can be done by creative programmers. The possibilities are limited only by the programmer's imagination, dedication, and free time.

PROGRAM IDEAS

Programming flexibility provided by shape tables enhances the instructional impact of computers. Shape tables can be applied in many ways to improve and expand educational programming packages for all disciplines.

Shape tables can provide the figures needed for intriguing games. Applesoft BASIC allows figures to be manipulated in an appropriate manner for nearly any game scheme.

Shape tables that contain the letters of the alphabet and a set of number symbols can be used in conjunction with other high-resolution graphics programming techniques to create many types of labeled graphs. Since the characters can be rotated, graph labels can be written vertically when necessary.

HELPFUL HINTS

Information presented in this section is not essential to creating shape tables, but it is helpful to be familiar with the following topics in order to utilize shape tables effectively:

1. Memory structure and selection.
2. Debugging and editing shape tables.
3. Indexing for table expansion.

Memory Structure and Selection. Selecting appropriate memory ranges for shape table placement is enhanced by understanding the structure and organization of computer memory. A brief explanation of the computer's memory structure is presented in Appendix D. If you are not familiar with Apple's memory, review that material now.

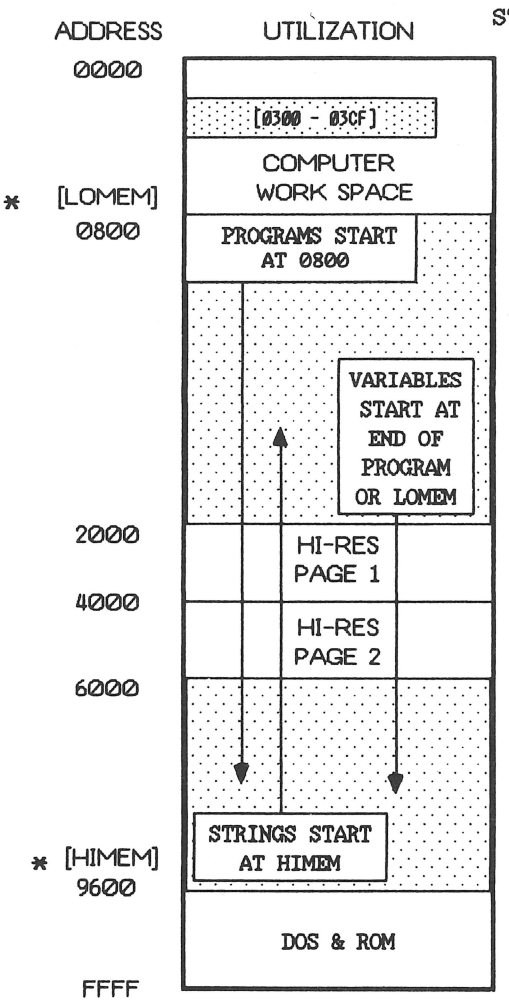
It is important that memory locations for shape tables be carefully selected. While there are several safe areas in memory for storing shape tables, it is helpful to select a range that meets personal programming requirements. Once that range has been selected, it should be used consistently while learning to design and use shape tables. For example, some programmers always use page 2 ($4000-$5FFF) for high-resolution graphics displays. Since page 1 ($2000-$3FFF) is not used for displaying graphics, shape tables can be safe stored in that memory range.

Figure 5-7 depicts the storage ranges of shape tables. The advantages and disadvantages of each range should be considered when selecting a place to store shape tables.

If none of the memory ranges shown in Fig. 5-7 are acceptable due to program size and graphics display requirements, then a more sophisticated approach to storing tables is necessary. The Apple stores string variable values from address $9600 down toward the graphics pages. You can alter the highest memory location used for string storage with the HIMEM: command and then store shape tables above the strings. Use of the Applesoft statement HIMEM:36864 would cause strings to be stored from address $8FFF down. This frees a large memory range from $9000 to $95FF for shape table storage. That's enough room to store a table with 1536 codes. The HIMEM statement should appear in the first line of the Applesoft program.

The computer always places a shape table back into the memory range from which it was saved (using BSAVE) unless a different range is specified when BLOAD brings it back from the disk. For example, if BSAVE SAMPLE SHAPES, A$3000, L$14A is used to store the shape table on disk, then BLOAD SAMPLE SHAPES is all that is necessary to place it back into memory range $3000 to $3149.

If SAMPLE SHAPES is to be loaded into the range $5000 to $5149, then the statement BLOAD SAMPLE SHAPES, A$5000 would be used. It is



Area 1

ADVANTAGES

1. Frees both graphics pages for displays.
2. Permits normal utilization of programming and variable storage memory ranges.

DISADVANTAGE

1. Shape Tables can only contain up to 208 hex codes.

Area 2

ADVANTAGES

1. Ample storage space for very large shape tables.
2. Room for several shape tables at one time.

DISADVANTAGES

1. Restricts graphics displays to page 2.
2. Long programs may extend into the area, clobbering the shape table.

Area 3

ADVANTAGES

1. Ample storage space for very large shape tables.
2. Room for several shape tables at one time.

DISADVANTAGES

1. Restricts graphics displays to page 1.
2. Strings or variables may extend into the area, clobbering the shape table.

Fig. 5-7. Shape tables can be placed anywhere in memory that is marked as free space, so long as they do not interfere with the program and its variables. Three of the safest areas for placing shape table are pictured in this figure.

not necessary to specify the length of a shape table when BLOADing it from the disk.

Debugging Shape Tables. Initial attempts at using shape tables occasionally result in unexpected video displays. This section can be used as a guide for identifying and correcting display problems. The usual problems are that the shapes do not appear on the screen, or that the displayed shapes do not appear as planned.

Shapes are not displayed on the screen if any of the following statements are true.

1. The shape table was not BLOADed from the disk.
2. The shape table was not saved on the disk being used.
3. The starting address of the shape table was not properly placed in memory locations $00E8 and $00E9 (232 and 233 decimal).
4. The background color and the color for drawing the shape are incompatible.
5. The index of the shape table was not properly compiled or entered.
6. A high-resolution graphics page is not displayed. (HGR or HGR2 must precede a DRAW command.)
7. The shape table was BLOADed into the memory range of the graphics page being used to display the shapes. (HGR and HGR2 completely erase their memory ranges.)

Shapes do not appear as planned if any of the following statements are true:

1. The index codes do not reference the first hex code of the shape definition.
2. The background color and the color being used for displaying the shapes are not compatible.
3. The coding was not done accurately for each step in the process. (One incorrect code will often result in considerable shape distortion.)
4. Each shape definition does not end with the code 00.

Once an error has been discovered, it can normally be corrected by returning to the Monitor, locating the appropriate shape definition, and retyping the errant codes. Occasionally errors are caused by omitting codes from the table or by entering a set of codes more than once.

Correction of these types of errors requires the use of the Monitor's *memory move* command, the syntax for which is:

{destination address}<{ab1}.{ab2}M

where ab1 is the address of the first byte of the segment to be moved and ab2 is the address of the last byte of the segment to be moved. For example, to move the contents of the memory segment $3000 through $3100 into the memory segment starting at $2000, the following command is used:

```
2000<3000.3100M
```

If codes have been omitted from the shape table, all of the codes that appear after the error must be moved. The missing code or codes must be entered, and then the moved segment must be relocated starting in the first memory byte after the corrections. For example, if the code OE in address $3012 had been omitted from the Sample Shapes shape table, the following sequence of instructions would have to be made from the Monitor:

```
6012<3012.3148M
3012:0E
3013<6012.6148M
```

The segment of memory used to store the moved codes temporarily must be far enough away from the shape table to make sure no overlapping occurs. It is best to select a starting address for the storage segment that has the same last three digits as the starting address of the segment from which it was moved. This makes it easier to determine the memory move parameters needed for placing the codes back into the appropriate area of the shape table.

Deleting a code or a segment of code from a

shape table requires even less effort than inserting missing codes. All that is required is simply to move the segment following the error to the correct memory location. For example if the code 3F at memory location $3019 had been duplicated in location $301A, the following statement would correct the error:

```
301A<301B.314AM
```

Additional information concerning the Monitor commands and their uses can be found in the Apple II Reference Manual.

Indexing for Table Expansion. The technique for shape table indexing discussed in Chapter 4 is the most efficient for memory utilization. If additional shape definitions must be placed in the table at a later time, the index can be extended and corrected by using the Monitor move command to make room for the additional index information required. When memory space is not a primary concern, however, or if a shape table is developed to which shapes may need to be added at a later date, then room should be reserved to accommodate the additional shape definitions.

Previous instructions in this chapter dictated that shape definitions begin in the first memory location following the index; in fact, however, they may begin anywhere beyond the last memory location used for indexing. This means that room can be left between the index and the first shape definition to be used for indexing additional shapes later. Place two 00 hex codes in the index for each shape expected to be added to the table. Remember that the maximum number of shapes allowed in a table is 255.

6

Shape Table Utility Programs

It should never be overlooked that the computer is a valuable tool for performing repetitive tasks. Creating shape tables and many of the action displays in which they are used is repetitive. The three utility programs presented in this chapter—Table Developer, Table Merger, and Shapely Scenes—are designed to make the job of utilizing shape tables in BASIC programs easier for the programmer.

TABLE DEVELOPER

Creating shape tables is very time-consuming when first learning the process. There are no substitutes for practice and experience in trying to decrease the time required to construct shape tables. One alternative, however, is to write a program that will do much of the work involved in shape table creation. Such a program, Table Developer, is developed in Listing 6-1.

The Table Developer program allows up to 255 shape definitions to be included in each shape table developed. The maximum number of vectors allowed for any shape is 3000, and the table capacity is from 12300 to 18400 vectors—or, more accurately, 6150 hexadecimal codes. The program can be used to create or extend any shape table. Rather than provide a line-by-line description of the program, let's take a look at the operating instructions for the program.

When the Table Developer program is run, instructions are presented that describe the program capacity and the commands used to create shapes. You are allowed either to create a new shape table or to extend an existing shape table. If option 2 (extending an existing table) is selected, you must provide the name of the shape table to be extended. Only tables created with the Table Developer program may be extended, and the disk containing the table must be a Drive 1 (or the default drive). The shape table is loaded into memory starting at $4000. This area was selected because high-resolution graphics page 1 is used to display the shapes being defined. After the table is loaded, Options 1 and 2 operate similarly.

Listing 6-1. Table Developer utility program (continues through page 112).

```
5  REM   TABLE DEVELOPER
6  LOMEM: 24576
10  TEXT : GOSUB 900: HOME : PRINT "OPTIONS ARE: ": PRINT
     : PRINT "1.  CREATE NEW SHAPE TABLE": PRINT : PRINT
     "2.  ADD SHAPES TO AN EXISTING TABLE": PRINT : PRINT
50  INPUT "ENTER 1 OR 2:  ";BR: IF BR < 1 OR BR > 2 THEN
     50
60  DIM BCZ(3200),SPZ(255): HOME :S$(0) = "OFF":S$(3) =
     "ON": HGR2 :FB = 2: FOR X = 0 TO 3: READ D$(X): NEXT
     X: TEXT : HOME : IF BR = 2 THEN  GOSUB 750
90  VTAB 22: HTAB 30: PRINT "BEGIN": VTAB 24: PRINT "C
     OMMANDS: I, J, K, M, S, ESC, <-";: VTAB 10: HGR :
     R = 0:C = 0:T = 3
100  GET G$: IF G$ =  CHR$ (13) OR G$ =  CHR$ (10) OR
     G$ = "" THEN 100
110  IF K = 2500 THEN  VTAB 22: PRINT "MAXIMUM VECTORS
      = 3000"
120  IF G$ <  > "S" THEN 129
121 K = K + 1: IF BU = 0 THEN BCZ(K) = BCZ(K - 1) + CC
      *  - 1
126  GOTO 500
129  IF K = 3000 THEN  PRINT  CHR$ (7); CHR$ (7); CHR$
     (7);: GOTO 100
130 G =  ASC (G$): IF G <  > 8 THEN 260
140 BU = 1:T = 0: IF K < 2 THEN  HTAB 1: VTAB 23: PRINT
     "RESTARTING": FOR Z = 1 TO 500: NEXT Z:K = 0: HOME
     : GOTO 90
160  FOR Z = 0 TO 3: IF BCZ(K - 1) = Z OR BCZ(K - 1) =
     Z + 4 THEN G$ = D$(Z): GOTO 175
170  NEXT Z
175  FOR Z = 0 TO 3: IF BCZ(K) = Z OR BCZ(K) = Z + 4 THEN
      HCOLOR= 0: HPLOT 140 + C,75 + R: HCOLOR= T:K = K
      - 1: ON Z + 1 GOTO 180,190,200,210
176  NEXT Z
180 R = R + 1: GOTO 220
190 C = C - 1: GOTO 220
200 R = R - 1: GOTO 220
210 C = C + 1
220 Q1 = 0: IF BCZ(K + 1) > 3 THEN Q1 = 3
230  HTAB 30: VTAB 22: PRINT S$(Q1)"  "G$" "K" ": VTAB
     10:MF = 0: IF Q1 = T THEN BU = 0: GOTO 100
240 MF = 4: GOTO 100
260  IF G <  > 27 THEN BU = 0: GOTO 280
261 T =  ABS (3 - T): IF BU = 0 THEN 267
262  IF T = 0 THEN 100
263  IF T = Q1 THEN MF = 0: GOTO 100
264 MF =  - 4: GOTO 100
267 MF = 4: IF T = 3 THEN MF =  - 4
268  GOTO 100
```

```
280  HTAB 30: VTAB 22: PRINT S$(T)"  "G$" "K + 1"  ": VTAB
     10: FOR X = 0 TO 3: IF G$ = D$(X) THEN K = K + 1:
      ON X + 1 GOTO 340,370,400,430
300  NEXT X: GOTO 100
340  IF R + 75 < 1 THEN 490
350 R = R - 1:BCZ(K) = X + MF + T * 1.5: GOSUB 460: GOTO
     100
370  IF C + 140 > 278 THEN 490
380 C = C + 1:BCZ(K) = X + MF + T * 1.5: GOSUB 460: GOTO
     100
400  IF R + 75 > 150 THEN 490
410 R = R + 1:BCZ(K) = X + MF + T * 1.5: GOSUB 460: GOTO
     100
430  IF C + 140 < 1 THEN 490
440 C = C - 1:BCZ(K) = X + MF + T * 1.5: GOSUB 460: GOTO
     100
460 CC = MF:MF = 0: HCOLOR= 3: HPLOT C + 140,R + 75: IF
     T = 0 THEN  FOR Z = 1 TO 35: NEXT Z
480  HCOLOR= T: HPLOT C + 140,R + 75: RETURN
490  VTAB 23: PRINT  CHR$ (7);: PRINT "ILLEGAL MOVE": FOR
     Z = 1 TO 200: NEXT Z: VTAB 23: PRINT  SPC( 38)" "
     :K = K - 1: VTAB 10: GOTO 100
500  HOME : VTAB 23: INPUT "DO YOU WANT THIS DESIGN IN
      THE TABLE?";G$: HOME : IF  LEFT$ (G$,1) = "N" THEN
     K = 0: GOTO 690
510 V = 1:S = 0: VTAB 23: HTAB 17: FLASH : PRINT "ENCO
     DING": NORMAL
511  FOR X = 2 TO K: IF BCZ(X) > 0 THEN FL = 0: GOTO 5
     13
512 FL = FL + 1: IF FL = 2 THEN 515
513  NEXT X
514  GOTO 540
515  IF BCZ(X + 1) > 0 AND BCZ(X + 1) < 4 THEN 513
516  FOR Y = K + 2 TO X + 3 STEP  - 1:BCZ(Y) = BCZ(Y -
     2): NEXT Y:BCZ(X + 1) = 1:BCZ(X + 2) = 3:K = K +
     2: GOTO 511
540 V = V + 1: IF V > K THEN S = S + 1: GOTO 600
550 S = S + 1: ON S GOTO 560,570,610
560 CHZ(S) = BCZ(V): GOTO 540
570  IF BCZ(V) <  > 0 THEN CHZ(S) = BCZ(V): GOTO 540
580  IF BCZ(V + 1) < 4 AND BCZ(V + 1) > 0 THEN CHZ(S) =
     BCZ(V): GOTO 540
590 CF = 1:CHZ(S) = 0:CHZ(S + 1) = 0: GOTO 630
600 CHZ(S) = 0:CHZ(S + 1) = 0: GOTO 630
610  IF BCZ(V) < 4 AND BCZ(V) > 0 THEN CHZ(S) = BCZ(V)
     : GOTO 630
620 CHZ(S) = 0:CF = 1
630 XZ = CHZ(1): IF  INT ((CHZ(2) - 1) / 2) = (CHZ(2) -
     1) / 2 THEN XZ = XZ + 8
640  POKE 16896 + HC, INT (CHZ(2) / 2 + CHZ(3) * 4) *
     16 + XZ:HC = HC + 1:S = 0: IF CF = 1 THEN CF = 0:
```

```
      GOTO 550
655   IF V >  = K THEN V = 0:K = 0:HC = HC + 1: GOTO 67
      0
660   GOTO 540
670 SH = SH + 1: POKE 16384,SH:SPZ(SH + 1) = HC
675   IF SPZ(SH + 1) > 255 THEN SPZ(SH + 1) = SPZ(SH +
      1) - 256: GOTO 675
680   POKE 16384 + 2 * SH,SPZ(SH): POKE 16385 + 2 * SH,
      FB:FB = 2 +  INT (HC / 256): IF SH = 255 THEN 710

685   IF HC + 16896 >  = 24575 THEN 710
690   HOME : VTAB 23: INPUT "ANOTHER DESIGN?  (Y/N)  ";
      G$
700   IF G$ = "N" THEN 705
701 X =  FRE (0): HOME : GOTO 90
705   HOME : VTAB 21: INPUT "WANT TO SAVE THIS TABLE?
      (Y/N)  ";G$: IF  LEFT$ (G$,1) <  > "N" THEN  HOME
      : GOTO 710
706   HOME : GOTO 730
710   VTAB 22: INPUT "NAME OF SHAPE TABLE? ";N$
720   PRINT  CHR$ (4)"BSAVE "N$",A$4000,L"512 + HC
730   TEXT : END
740   DATA    "I","K","M","J"
750   HOME : PRINT "ENTER NAME OF TABLE TO BE EXTENDED:
      ": PRINT : INPUT N$
760   PRINT  CHR$ (4)"BLOAD"N$
770 SH =  PEEK (16384):FB =  PEEK (16384 + 2 * SH + 1)
      :SPZ(SH) =  PEEK (16384 + 2 * SH):K = 16384 + FB *
      256 + SPZ(SH)
810 V = V + 1: IF  PEEK (K + V) = 0 THEN HC = K + V -
      16895: GOTO 840
830   GOTO 810
840   PRINT : PRINT : PRINT "CURRENT # OF SHAPES = "SH:
       VTAB 23: PRINT "PRESS ANY KEY TO BEGIN ADDING";:
       GET G$: HOME :SPZ(SH + 1) = HC
860   IF SPZ(SH + 1) > 255 THEN SPZ(SH + 1) = SPZ(SH +
      1) - 256: GOTO 860
870 FB = 2 +  INT (HC / 256):K = 0:V = 0: RETURN
900   HOME : PRINT "MAXIMUM VECTORS IN DESIGN = 3000": PRINT
      : PRINT "MAXIMUM NUMBER OF SHAPES = 255": PRINT :
       PRINT "TABLE CAPACITY IS FROM 12300 TO 18400   V
      ECTORS CODES."
910   PRINT : PRINT : PRINT "COMMANDS ARE:  I...UP"
915   PRINT : PRINT "                J...LEFT": PRINT
916   PRINT "             K...RIGHT": PRINT
917   PRINT "             M...DOWN": PRINT : PRINT "(
      LEFT ARROW)  <-...DELETE VECTOR"
918   PRINT : PRINT "                S...STOP"
919   PRINT : PRINT "           ESC...PLOT/NO PLOT": VTAB
      23: PRINT "PRESS A KEY TO CONTINUE.";: GET G$: HOME
922   PRINT "    SHAPES WILL BE DISPLAYED IN THE": PRINT
```

```
     : PRINT "CENTER OF THE SCREEN.  WHEN THE WORD": PRINT
     : PRINT "'BEGIN' IS DISPLAYED, THE 'PLOT' MODE": PRINT

     : PRINT "IS ENGAGED.  YOU MAY SWITCH TO THE 'NO":
      PRINT : PRINT "PLOT' MODE BY PRESSING THE ESCAPE
      (ESC)"
924  PRINT : PRINT "KEY.  THE PLOTTING MODE CHANGES EA
     CH": PRINT : PRINT "TIME THE ESC KEY IS PRESSED."
     : PRINT
925  PRINT "    WHEN LEFT ARROW (BACK SPACE) IS": PRINT
     : PRINT "USED TO DELETE VECTORS, THE 'NO PLOT'": PRINT

     : PRINT "MODE IS AUTOMATICALLY ENGAGED. PRESS THE
     ": PRINT "ESCAPE KEY TO RESTORE THE 'PLOT' MODE."

930  VTAB 24: PRINT "     PRESS ANY KEY TO CONTINUE.";
     : GET G$: HOME : RETURN
```

High-resolution page 1 is cleared and displayed. At the bottom of the screen is a list of the keys used to direct the formation of a shape. The word BEGIN also appears to indicate the program is ready to receive commands. The plot status is constantly displayed at the bottom of the screen. If the word ON is displayed, the next vector plotted will be visible; OFF means the next vector will not be visible. In addition, the number of vectors plotted and the lask key pressed are also displayed. Designs are created by pressing R, L, U, and D to move around the screen. The ESC key is used to switch between plotting and nonplotting vectors. The left arrow key is used to delete vectors. The nonplotting mode is invoked whenever the left arrow key is pressed. S, for stop, is used to signal the end of a design.

The Table Developer program allows you to move upward on the screen as many times as desired in the nonplotting mode, but more efficient coding results if such moves are avoided. Whenever you try to move off of the screen, the message ILLEGAL MOVE is displayed and the bell is sounded. The bell is sounded three times if you try to use more than 3000 codes in a single design.

Once the display on the screen forms an acceptable shape for the table, the S key should be pressed. The program asks if the shape is to be placed in the table. Typing Y causes the shape to be placed in the table. (A delay of approximately 13) seconds for each 100 vectors in the shape occurs while the program encodes the vectors and places them in the table.) If N is pressed, the shape is not placed in the table. In either case, the program asks if another shape is to be designed. If Y is pressed the designing process starts over for the next shape; after an N response, the program asks if the shape table is to be saved on disk. Answering N to this question causes the program to terminate without saving the shape table. If the question is answered by typing Y, you are asked to name the table. Remember that all filenames must begin with a letter and may not contain commas. The program ends after the table is saved.

All shape tables created with the Table Developer program are saved from memory starting at $4000. If the program that displays the shapes expects to store the table in a different area of memory, the address parameter must be specified with the BLOAD command used to place the table in memory.

Although the program makes shape table development easier, some of you may find it is more difficult to determine the relative starting point of a shape, and even more difficult to make certain that the starting points of sets of shapes are the same. This is an extremely important consideration when trying to develop a shape table like Character Set,

since it is essential to have the same relative starting point for all shapes in the table. Even when using the Table Developer program, better shape tables are designed if you first construct the desired shapes on graph paper and trace the vector paths by hand.

TABLE MERGER

There are times when a set of shape definitions from one table needs to be added to another table. This tedious task can also be made easier by using the computer to extract the shapes from one table, place them in another, and finally modify the index of the receiving table. A program, Table Merger, that accomplishes these functions is shown in Listing 6-2. The program can only be used with tables that have been created with the Table Developer program.

Listing 6-2. Table Merger utility program.

```
2  REM   TABLE MERGER
5  HGR2 : TEXT : HOME
6  INPUT "NAME OF DONOR TABLE: ";TD$
10  PRINT  CHR$ (4)"BLOAD"TD$",A$6000"
11  INPUT "NAME OF RECEIVING TABLE: ";RT$
14  DEF  FN DIV(RA) =  INT (RA / 256)
15  DEF  FN MOD(RA) =  INT ((RA / 256 -  INT (RA / 256
     )) * 256)
16 NS =  INT (4 * 16 ^ 3)
17 SS =  INT (4 * 16 ^ 3 + 2 * 16 ^ 2)
18 OT =  INT (6 * 16 ^ 3)
19  GOSUB 1000
20  POKE 233,96: POKE 232,0
21  HGR : VTAB 21
22  HCOLOR= 3
25  PRINT "PRESS <RETURN> TO QUIT."
30  INPUT "ENTER A SHAPE NUMBER: ";K$
35  IF  NOT  VAL (K$) THEN 500
40 K =  VAL (K$)
50  IF K >  PEEK (OT) OR K < 1 THEN 30
51  HGR : DRAW K AT 140,70
52  INPUT "CONTINUE (Y/N)? ";G$: IF G$ <  > "Y" AND G$
     <  > "N" THEN 52
55  IF G$ = "N" THEN 30
60 OL = OT + K * 2
70 LO =  PEEK (OL):HO =  PEEK (OL + 1)
71  POKE NS, PEEK (NS) + 1
72  POKE NS +  PEEK (NS) * 2, FN MOD(SS - NS): POKE NS
     +  PEEK (NS) * 2 + 1, FN DIV(SS - NS)
75 SO = LO + HO * 256 + OT
80  POKE SS, PEEK (SO)
90  IF  NOT  PEEK (SO) THEN 200
100 SS = SS + 1:SO = SO + 1
110  GOTO 80
200 SS = SS + 1:SO = SO + 1: GOTO 30
500  INPUT "NAME OF NEW TABLE? ";TB$
510  PRINT  CHR$ (4)"BSAVE"TB$",A"NS",L"SS - NS + 1
```

```
520  TEXT : HOME : END
1000  PRINT  CHR$ (4)"BLOAD"RT$",A$4000"
1010 V =  PEEK (NS)
1020 Z =  PEEK (NS + 2 * V) +  PEEK (NS + 2 * V + 1) *
     256 + NS
1030  IF  PEEK (Z) THEN Z = Z + 1: GOTO 1030
1040 SS = Z + 1
1050  RETURN
```

The program first asks for the name of the donor shape table, which is the one from which shapes are to be copied. Next, the program asks the name of the receiving table. The shape definitions copied from the donor table are appended to the receiving table in the order they are selected. Once the tables have been identified, each shape that is to be transferred must be identified, one at a time. After a shape number has been selected, the program displays the shape on the screen and asks if it should continue. If the response to this question is Y, the shape is appended to the receiving table. If the response is N, the program asks for another shape number to be specified. Pressing <Return> in response to the request for a shape number informs the program that no more shapes are to be transferred. The program then asks for a name for the newly formed shape table and saves it under that name on the disk.

Once figures have been defined in shape tables, they become a permanent resource for graphics displays. Having an extensive repertory of designs stored in shape tables frees time for creating programs which display graphics that may have otherwise been spent simply programming high-resolution graphics designs. Other programming approaches can further expedite software development.

SHAPELY SCENES

The Shapely Scenes program (Listing 6-3) allows the formation of graphics displays using figures stored in shape tables. It can even be used to create action scenes. The program allows the user to specify the shape—and control the color, rotation, exposure time, and screen position of the shapes—to create the desired graphics scene. When entering the program, the REM statements can be omitted without affecting the execution of the program.

Listing 3-6. Shapely Scenes (continues through page 118).

```
10  REM      SHAPELY SCENES
20  REM
30  REM  --------------------
40  REM  LOAD SHAPE TABLE, SCENE DATA FILE
50  REM  INTIALIZE VARIABLES FROM DATA
60  REM  CLEAR GRAPHICS PAGES
70  REM  --------------------
80  REM
90 D$ =  CHR$ (4):SA = 16384
100  HOME : INPUT "NAME OF SHAPE TABLE? ";ST$
110  PRINT D$"BLOAD"ST$",A$6000"
120  POKE 232,0: POKE 233,96: SCALE= 1: ROT= 0
130 C = 140:R = 80
140  FOR X = 1 TO 9: READ M$(X): NEXT : FOR X = 1 TO 7
     : READ C$(X): NEXT : FOR X = 1 TO 5: READ E$(X): NEXT
```

```
150  HGR2
160  HGR
170  HTAB 1: VTAB 23: INPUT "NAME OF SCENE: ";SD$
180  ONERR  GOTO 220
190  PRINT D$"BLOAD"SD$
200  POKE 216,0
210  GOSUB 9000
220  POKE 216,0
230  REM
240  REM  ----------------------
250  REM  POSITION SHAPE ON SCREEN
260  REM  ----------------------
270  REM
280  GOSUB 800
290  IF  NOT ST THEN ST = 1
300  HCOLOR= 3: DRAW SH AT C,R
310  VTAB 5: GET MOVE$: FOR X = 1 TO 9: IF MOVE$ = M$(
     X) THEN 340
320  NEXT X: IF  VAL (MOVE$) THEN ST =  VAL (MOVE$)
330  GOTO 310
340  HCOLOR= 0: DRAW SH AT C,R: ON X GOSUB 370,410,450
     ,510,660,560,590,590,490
350  GOTO 290
360  REM   ** MOVE SHAPE UP (ST) ROWS **
370  IF R - ST < 1 THEN  RETURN
380  HCOLOR= 0: DRAW SH AT C,R
390 R = R - ST: RETURN
400  REM  ** MOVE SHAPE LEFT (ST) COLUMNS **
410  IF C - ST < 1 THEN  RETURN
420  HCOLOR= 0: DRAW SH AT C,R
430 C = C - ST: RETURN
440  REM  ** MOVE SHAPE RIGHT (ST) COLUMNS **
450  IF C + ST > 255 THEN  RETURN
460  HCOLOR= 0: DRAW SH AT C,R
470 C = C + ST: RETURN
480  REM  ** CENTER SHAPE ON SCREEN **
490  HCOLOR= 0: DRAW SH AT C,R:C = 140:R = 70: RETURN
500  REM  ** MOVE SHAPE DOWN (ST) ROWS **
510  IF R + ST > 159 THEN  RETURN
520  DRAW SH AT C,R
530 R = R + ST: RETURN
540  HCOLOR= 0: DRAW SH AT C,R
550  REM  ** ROTATE SHAPE RIGHT 45 DEGREES **
560  IF RO > 247 THEN  RETURN
570  HCOLOR= 0: DRAW SH AT C,R:RO = RO + 8: ROT= RO: RETURN
580  REM  ** ROTATE SHAPE LEFT 45 DEGREES **
590  IF RO < 8 THEN  RETURN
600  HCOLOR= 0: DRAW SH AT C,R:RO = RO - 8: RETURN
610  REM
620  REM ----------------------
```

```
630  REM  STORE SHAPE SPECS IN SCENE DATA TABLE
640  REM ------------------------
650  REM
660 DP = SA +  PEEK (SA) * 7 + 1
670  POKE SA, PEEK (SA) + 1
680  HOME : VTAB 23: INPUT "COLOR,TIME: ";CO,TI
690  POKE DP,SH:DP = DP + 1: POKE DP,CO:DP = DP + 1
700  POKE DP,RO:DP = DP + 1
710  POKE DP,SC:DP = DP + 1
720  POKE DP,C:DP = DP + 1: POKE DP,R:DP = DP + 1: POKE
     DP,TI
730  HCOLOR= CO: DRAW SH AT C,R
740  IF ED THEN  RETURN
750  REM
760  REM -------------------------
770  REM OBTAIN MAIN OPTION SELECTION
780  REM -------------------------
790  REM
800  HTAB 1: VTAB 21: CALL  - 958: PRINT
810  PRINT "<R>ECALL  <E>DIT  <A>DD  <Q>UIT ";
820  GET G$: IF G$ = "R" AND  PEEK (SA) THEN  GOSUB 90
     00: GOTO 800
830  IF G$ = "A" THEN  GOTO 870
840  IF G$ = "E" THEN  GOSUB 1120: GOTO 800
850  IF G$ = "Q" THEN  GOTO 1490
860  GOTO 820
870  PRINT
880  INPUT "SHAPE, SCALE:";SH,SC: IF SH < 1 OR SH >  PEEK
     (24576) THEN 880
890  HCOLOR= 3: SCALE= SC
900  GOSUB 1040
910  RETURN
920  REM
930  REM  ----------------------
940  REM   DISPLAY PLACEMENT (MOVE) PROMPTS
950  REM   <I>  UP       <M>  DOWN
960  REM   <J>  LEFT     <K>  RIGHT
970  REM   <R> ROTATE RIGHT
980  REM   <L> ROTATE LEFT
990  REM   <P> PLACE IN CURRENT POSITION
1000  REM   <Q> QUIT - REMOVE SHAPE
1010  REM   <C> PLACE IN CENTER OF SCREEN
1020  REM  ----------------------
1030  REM
1040  HOME : VTAB 23: PRINT "<I> <J> <K> <M> <P> <R> <
     L> <Q> <C>"
1050  RETURN
1060  REM
1070  REM -------------------------
1080  REM  DISPLAY A SHAPE FROM THE SCENE
1090  REM   THEN OBTAIN EDITING INSTRUCTIONS
```

```
1100  REM ------------------------
1110  REM
1120  HGR : IF  NOT  PEEK (SA) THEN ED = 0: RETURN
1130 X = SA - 6:ED = 1
1140 X = X + 7
1150  HCOLOR=  PEEK (X + 1): ROT=  PEEK (X + 2): SCALE=
      PEEK (X + 3): DRAW  PEEK (X) AT  PEEK (X + 4), PEEK
     (X + 5)
1160  HTAB 1: VTAB 21: CALL  - 958: FOR Y = 0 TO 3
1170  PRINT C$(Y + 1) PEEK (X + Y)" ";: NEXT Y: PRINT
1180  FOR Y = 4 TO 6: PRINT C$(Y + 1) PEEK (X + Y)"  "
     ;: NEXT Y: PRINT
1190  PRINT "<E>DIT <C>ONT <B>ACK <I>NSERT <D>ELETE:";
1200  GET G$
1210  FOR Z = 1 TO 5: IF G$ = E$(Z) THEN 1230
1220  NEXT Z: GOTO 1200
1230  ON Z GOTO 1240,1340,1360,1410,1390
1240 DP = X:C =  PEEK (X + 4):R =  PEEK (X + 5):RO =  PEEK
     (X + 2):CO =  PEEK (X + 1):SC =  PEEK (X + 3):TI =
      PEEK (X + 6):SH =  PEEK (X)
1250  PRINT : INPUT "SHAPE, SCALE:";SH,SC: IF SH < 1 OR
     SH >  PEEK (24576) THEN 1250
1260  VTAB 21: CALL  - 958: GOSUB 1040: SCALE= SC:ST =
     1
1270  HCOLOR= 3: DRAW SH AT C,R
1280  VTAB 5: GET MOVE$: FOR Z = 1 TO 9: IF MOVE$ = M$
     (Z) THEN 1300
1290  NEXT Z: IF  VAL (MOVE$) THEN ST =  VAL (MOVE$)
1300  HCOLOR= 0: DRAW SH AT C,R: ON Z GOSUB 370,410,45
     0,510,1320,560,590
1310  GOTO 1270
1320  GOSUB 680
1330 X = DP - 6: POP : GOTO 1150
1340  IF (X + 7 >  PEEK (SA) * 7 + SA) THEN ED = 0: RETURN

1350  GOTO 1140
1360  IF X > SA + 7 THEN X = X - 7
1370  GOTO 1150
1380  REM   ** DELETE A SHAPE FROM SCENE DATA TABLE **

1390  FOR Z = X TO SA +  PEEK (SA) * 7: POKE Z, PEEK (
     Z + 7): NEXT Z: POKE SA, PEEK (SA) - 1: GOTO 1120

1400  REM   ** INSERT A SHAPE IN SCENE DATA TABLE **
1410  POKE SA, PEEK (SA) + 1: FOR Z = SA - 1 +  PEEK (
     SA) * 7 TO X + 7 STEP  - 1: POKE Z, PEEK (Z - 7)
1420  NEXT Z: GOTO 1240
1430  REM
1440  REM  ---------------------
1450  REM  OBTAIN NAME FOR SCENE DATA FILE
1460  REM  AND SAVE ON DISK
```

```
1470  REM  ----------------------
1480  REM
1490  PRINT : INPUT "NAME OF SCENE: ";G$: IF  LEN (G$)
      THEN SD$ = G$
1500  PRINT D$"BSAVE"SD$",A$4000,L" PEEK (SA) * 7 + 1:
      TEXT : END
1510  REM
1520  REM  ----------------------
1530  REM   DISPLAY ENTIRE SCENE
1540  REM  ----------------------
1550  REM
9000  HGR : FOR X = SA + 1 TO  PEEK (SA) * 7 + SA - 1 STEP
     7: HCOLOR=  PEEK (X + 1): ROT=  PEEK (X + 2): SCALE=
      PEEK (X + 3): DRAW  PEEK (X) AT  PEEK (X + 4), PEEK
     (X + 5)
9010  FOR Y = 1 TO 4 *  PEEK (X + 6): NEXT Y,X: RETURN
9900  REM
9910  REM  ----------------------
9920  REM          DATA
9930  REM
9940  REM  10000: MOVE KEYS
9950  REM  10010: DATA DISCRIPTORS
9960  REM  10020: EDITING OPTIONS
9970  REM  ----------------------
9980  REM
10000  DATA  I,J,K,M,P,R,L,Q,C
10010  DATA    "SHAPE: ","COLOR: ","ROT: ","SCALE: ","
     COLUMN: ","ROW: ","TIME: "
10020  DATA  E,C,B,I,D
10030  REM
10040  REM  ----------------------
10050  REM    VARIABLE ROLES
10060  REM
10070  REM   D$: DOS COMMAND SIGNAL
10080  REM   SA: STARTING ADDRESS OF SCENE DATA
10090  REM   ST$: NAME OF SHAPE TABLE
10100  REM    C: COLUMN COORDINATE OF SHAPE
10110  REM    R: ROW COORDINATE OF SHAPE
10120  REM   M$(): ARRAY FOR MOVE COMMANDS
10130  REM   C$(): SHAPE SPECIFICATION TITLES
10140  REM   E$(): EDIT CHOICES ARRAY
10150  REM   SD$: SCENE DATA FILE NAME
10160  REM   ST: SKIP TAB FOR MOVING SHAPES
10170  REM   SH: SHAPE NUMBER
10180  REM   MOVE$: MOVE SELECTION
10190  REM   DP: SCENE DATA POSITION POINTER
10200  REM   CO: COLOR FOR DISPLAYING SHAPE
10210  REM   TI: EXPOSURE TIME
10220  REM   RO: ROTATION OF SHAPE
10230  REM   SC: SCALE OF SHAPE
10240  REM   ED: EDIT FLAG (1=EDIT) (0=NOT EDITING)
```

This program can be used with all shape tables, not just those created with the Table Developer program. A brief explanation of the roles of the variables and the functions of the major modules of the program is provided in REM statements in the program listing. The operating instructions and some of the programming strategies used are presented in the following paragraphs.

All the shapes needed for creating a scene must be in one shape table, and the disk containing that shape table must be in the drive when using the Shapely Scenes program. When the program is run, it first asks for the name of the shape table to be used. The next question requests the name of the scene file. Either an existing scene file or the name of the file that is to be created can be specified. Any legal filename can be used for a scene data file. If an old scene file is named, the scene is displayed on the screen. The program then displays the main option prompts

```
<R>ECALL <E>DIT <A>DD
<Q>UIT
```

at the bottom of the screen. Pressing the enclosed prompt letter causes the proper branch to be taken. It is not necessary to press the Return key after selecting an option.

The Add option should be selected whenever another shape is to be placed on the screen. The program will ask for the shape number and the scale value that is to be used for displaying the shape. The shape is then displayed on the screen. If the shape being added is the first to be added since the program was run, it is displayed at the center of the screen. If other shapes have been displayed, the current shape will be drawn at the coordinates used for displaying the previously positioned shape. The move prompts

```
<I>  <J>  <K>  <M>  <P>  <R>
<L>  <Q>  <C>
```

are then shown at the bottom of the screen.

The I, J, K, and M keys are used to move the shape up, left, right and down respectively. The shapes are normally moved one screen position at a time. However, the *move step*, or number of positions a shape is moved, can be altered by pressing one of the number keys. The move step then becomes the number corresponding to the pressed key. Pressing C causes the shape to be moved to the center of the screen. R and L are used to rotate the shape 45 degrees to the right or left respectively.

Pressing P or Q results in a branch back to the main options level. If Q is pressed, the shape is removed from the screen and does not become a part of the scene. If P is pressed, the program asks for the color and time parameters that are to be used in displaying the shape. The color codes 0 through 7 must be used to specify the color. Any number from 0 to 255 may be used to specify the time parameter. The time parameter determines the length of the delay that occurs before the next shape in the scene is displayed. Once all the necessary information is obtained, it is stored in memory and becomes a part of the scene.

The Recall main option causes the scene to be displayed on the screen. As shapes are moved around on the screen, previously positioned shapes may be either partially or completely removed from the screen. When this occurs, the R key should be pressed when the main option prompts are displayed.

Any scene may be modified by selecting the Edit main option. When E is pressed, the scene is displayed, one shape at a time. After a shape is displayed, all the parameters used for displaying the shape and the editing prompts

```
<E>DIT <C>ONT <B>ACK <I>NSERT
<D>ELETE
```

are shown at the bottom of the screen. The desired editing option may be selected by pressing the key that corresponds to the enclosed letter of the command. The Edit option allows the user to modify the shape number, scale, position, color, rotation, and time delay parameters. Pressing C causes the next shape in the scene to be displayed. When B is pressed, the parameters for the previous shape of the scene are presented, and the editing options

reappear. Pressing I permits another shape to be inserted into the scene just ahead of the currently displayed shape. The Delete option removes the currently displayed shape from the scene. After the entire scene has been stepped through one shape at a time, the program branches back to the main option level.

Selecting the last main option, Quit, causes the program to ask for the filename that is to be used for saving the screen file on the disk. Pressing the return key causes the filename specified at the beginning of the program to be used for saving the scene data file.

Some practice with the Shapely Scenes program may be required for many of you before all of its capabilities are realized. The principles discussed in the section on motion can be applied when designing scenes. By designing shapes that represent successive frames of a movement, action sequences can be simulated. Remember that a shape can be removed from the scene by redrawing it in the same position using the color black. The Shapely Scenes program does not allow specifying a background color, so black (0) is always used.

While it is not necessary to understand how the Shapely Scenes program works, it is helpful to understand how the scene data is organized. As the shape number and related information is obtained by the program, it is stored in the segment of memory normally used by the computer for page 2 high-resolution graphics displays. The first byte of the segment stores the number of shapes used in creating the scene. This means that only 255 shapes can be used in each scene. Seven bytes of memory are used to store the information needed to display each shape. The information is stored in this order: shape number, color, rotation, scale, column coordinate, horizontal coordinate, and time delay. Since only one byte is reserved for the column coordinate, no shapes can be positioned to the right of column number 255 of the screen. Users who find this restriction objectionable can modify the program so that two bytes of memory are reserved for each shape's column coordinate.

When the user selects the Quit main option, the program obtains a name for the scene, then BSAVEs it on the disk. The information could just as easily have been stored in arrays and then saved in a sequential text file on the disk. There are several advantages to using the binary file scheme, not the least of which is speed. Information can be accessed by a BASIC program from binary files much more quickly than from text files. Memory conservation is another area where the binary data organization wins out over the array alternative. Each element of an integer array uses two bytes of memory. In addition, the array has another seven bytes of overhead. Therefore, less than half the memory is required if data are stored in binary form. Fewer program statements are required to retrieve the information from a binary file than from a text file.

One disadvantage of using the binary data scheme, however, is that the programmer has to determine which memory segment should be used for storing data—but doing so is a small price to pay for all the advantages gained by using binary data.

Once a scene is developed using the Shapely Scenes program, it can be delivered quite easily from another program. The Scene Viewer program (Listing 6-4) demonstrates how to use the scenes in other programs.

Listing 6-4. Scene Viewer.

```
5  REM     SCENE VIEWER
6  REM
10 D$ =  CHR$ (4):SA = 16384
20  VTAB 14: INPUT "NAME OF SHAPE TABLE? ";ST$
26  PRINT : INPUT "NAME OF SCENE? ";SD$
30  PRINT D$"BLOAD"ST$",A$6000"
35  PRINT D$"BLOAD"SD$
40  POKE 232,0: POKE 233,96: SCALE= 1: ROT= 0
```

```
50  GOSUB 9000: GET G$: TEXT : HOME : END
9000  HGR : FOR X = SA + 1 TO  PEEK (SA) * 7 + SA - 1 STEP
     7: HCOLOR=  PEEK (X + 1): ROT=  PEEK (X + 2)
9010  SCALE=  PEEK (X + 3): DRAW  PEEK (X) AT  PEEK (X
     + 4), PEEK (X + 5): FOR Y = 1 TO 4 *  PEEK (X +
     6): NEXT Y: NEXT X
9020  RETURN
```

Notice that the subroutine at line 9000 is the same subroutine used in the Shapely Scenes program when the Recall main option is selected. The entire scene can be displayed anytime after the shape table and scene data files are loaded with the statement GOSUB 9000.

In order to illustrate the kinds of displays that can be achieved using the Shapely Scenes and Scene Viewer programs, a short scene data segment is presented in Listing 6-5. Enter the Monitor by typing CALL –151. The data should be placed in memory starting at $4000, so type 4000: next to the asterisk prompt character. Then enter the hex codes in the list, separated by spaces. After the codes have been entered, they can be stored in a binary file by using the command BSAVE LETTER SCENE, A$4000, L$135.

After the Letter Scene data are saved on the

Listing 6-5. Hexadecimal listing of Letter Scene shape table.

```
4000:2C 21 03 00 01 71 10 00 22 03 00 01
8C 10 00 23 03 00 01 A7 10 00 3B 03 20
0F 8B 50 FF 3B 00 20 0F 8B 50 00 3B 03
28 0F 58 55 00 23 00 00 01 A7 10 00 23
03 00 01 A7 22 00 22 00 00 01 8C 10

:00 22 03 00 01 8C 1D 00 21 00 00 01 71
10 00 21 03 10 01 7B 19 00 23 00 00 01
A7 22 00 23 03 00 01 A7 3D 00 22 00 00
01 8C 1D 00 22 03 00 01 8C 2B 00 21 00
10 01 7B 19 00 21 03 20 01 84 24 00 22

:00 00 01 8C 2B 00 22 03 10 01 9E 2C 00
23 00 00 01 A7 3D 00 23 03 00 01 A7 4F
00 22 00 10 01 9E 2C 00 22 03 20 01 B4
32 00 21 00 20 01 84 24 00 21 03 30 01
91 31 00 22 00 20 01 B4 32 00 22 03 20

:01 C1 3B 00 21 00 30 01 91 31 00 21 03
00 10 9A 39 00 22 00 20 01 C1 3B 00 22
03 30 01 CA 48 00 21 00 00 01 9A 39 00
21 03 10 01 A7 43 00 22 00 30 01 CA 48
00 22 03 00 01 CA 4F 00 21 00 10 01 A7

:43 00 21 03 20 01 AA 49 0A 21 00 20 01
AA 49 00 21 03 30 01 B2 4B 0A 21 00 30
01 B2 4B 00 21 03 00 01 B9 4F 00 3B 00
68 0F 58 55 00 3B 03 00 0F B8 17 FF
```

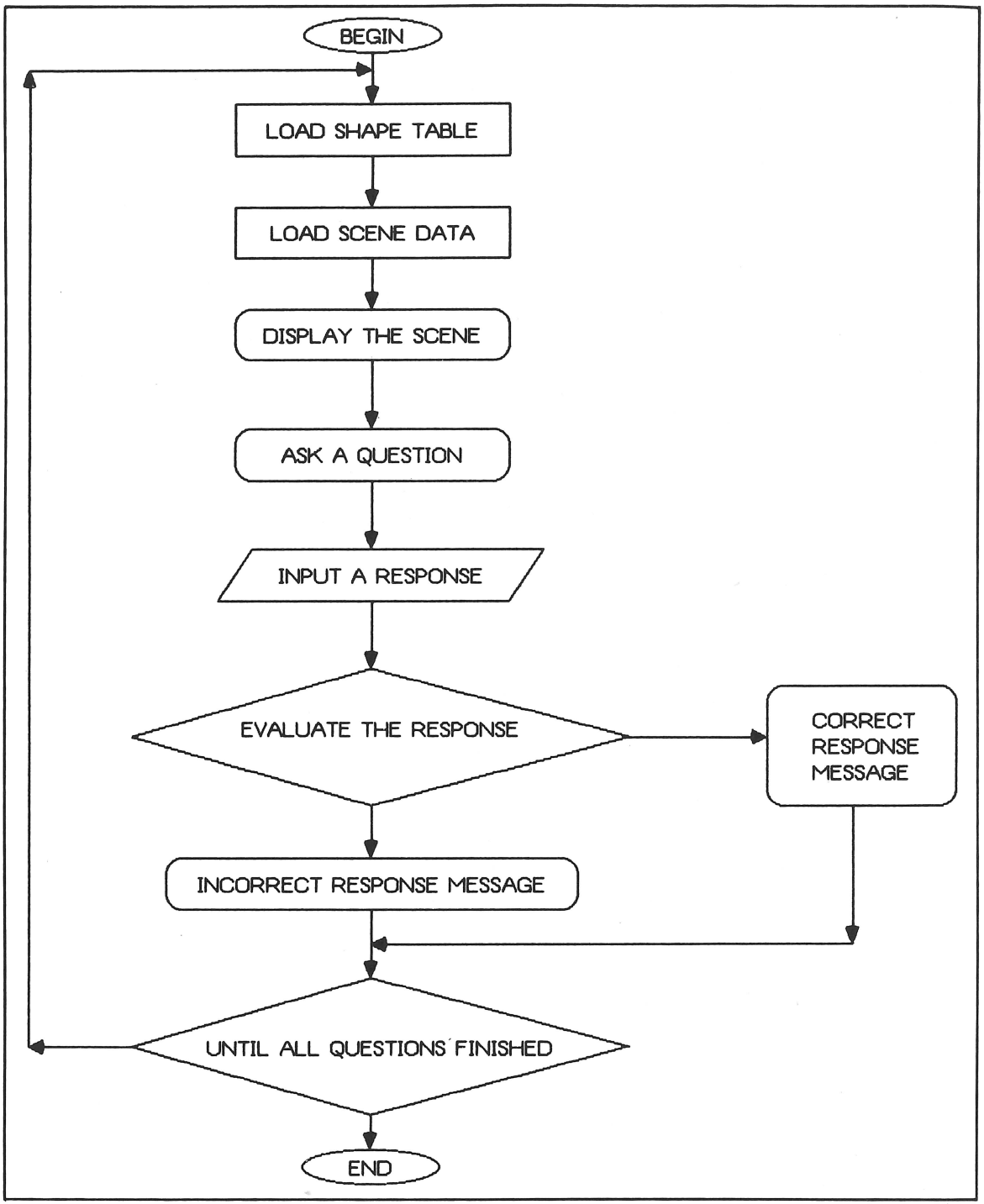


Fig. 6-1. A flowchart describing a program sequence that could be utilized with scenes developed using the programs in this chapter.

disk, either Shapely Scenes or Scene Viewer can be used to display the scene.

Graphics displays developed using the Shapely Scenes program are ideal for games or educational programs. They are particularly suitable for programs that progress from one pictorial display to another based upon the user's response to a question. These types of programs can be either entertaining, educational, or both; the time required to BLOAD the scene data for the next question does not distract from the program. No programs using the technique are presented in this book, but a programming sequence for developing programs is illustrated by the flowchart in Fig. 6-1.

7

Music and Sound

Music and other sounds are often used to enhance programs. Unfortunately, Applesoft BASIC has no commands that allow musical sounds to be generated easily. Routines that generate more sophisticated sounds than those discussed in Chapter 2 must be written in machine language. Once machine language programs that produce sounds have been developed, however, they can be utilized easily within BASIC programs.

Several machine language programs for producing sounds are presented in this chapter, but no attempt is made to explain them. Instructions for operating and utilizing these machine language routines from within BASIC programs are presented, however, to let you create simple sounds, music and songs, and arcade game sounds.

SIMPLE SOUNDS

Although we do not expect you to know machine language programming, some fundamental concepts must be addressed. All machine language commands or instructions are eight-digit binary numbers, which are very difficult for humans to handle. The computer allows the instructions to be entered as two-digit hex numbers, which makes the task much easier. However, machine language programs are generally developed with an Assembler program that allows machine language instructions to be entered using *opcodes*—which are meaningful three-letter symbols. The machine language listings presented in this book are given in assembled form as well as hexadecimal form. If you do not have an assembler, you can enter the hex programs from the monitor.

A popular technique for sound generation is to develop a machine language sound program that can play one note. The sound produced by the program is determined by two note values, pitch and duration. The note values are passed to the machine language sound program from within the BASIC program using POKE statements. The BASIC program then CALLs the machine language program, causing the note to be played. The Note Player program (Listing 7-1) is a machine language program that produces a sound each time it is executed.

Listing 7-1. Assembly language listing of Note Player program.

```
1                ORG   $303
2    PITCH       EQU   $300
3    LENGTH      EQU   $301
4    DELAY       EQU   $302
5    SPEAKER     EQU   $C030
6
7    *       NOTE PLAYER
8
9                LDA   #$00
10               STA   DELAY
11               LDY   LENGTH
12   START       LDX   PITCH
13   TWEAK       LDA   SPEAKER
14   WAIT        DEC   DELAY
15               BNE   TESTPCH
16               DEY
17               BEQ   END
18   TESTPCH     DEX
19               BEQ   START
20               BNE   WAIT
21   END         RTS
```

The hexadecimal listing of Note Player (Listing 7-2) should be entered from the Monitor, starting at $303. After entering the program, save it on the disk using the following statement:

BSAVE NOTE PLAYER, A$303, L28.

The computer does not have a built-in assembler, but a *disassembler* does reside in the Apple Monitor. After the hexadecimal listing has been entered from the Monitor, type 303L. The Monitor causes a listing that is similar to the assembly listing of the program to be displayed. The disassembler can be useful in determining if the hexadecimal listing has been correctly entered. Comparing the disassembled list to the assembler listing can help you detect typing errors.

Once the Note Player program has been entered, it can be executed from the Apple Monitor by typing 303G. The pitch of the note the program plays depends on the value in memory location $300, and the duration of the note depends on the value stored in memory location $301. Enter the following statements from the Apple Monitor to determine how changes in the note values affect the sound produced by the program.

```
300:A0 A0
303G
301:20
303G
300:20
303G
301:A0
303G
```

The BASIC program Note Tester (Listing 7-3) is equivalent to the series of Monitor commands listed above. The Note Player program must be located in memory at $303 before Note Tester can be run.

Memory location 768 is equivalent to $300. Note Player uses the value in this location to determine how frequently the speaker is to be "tweaked." The greater the value stored in $300, the lower the pitch of the note. The value stored in memory location 769 ($301) is used by the Note Player program to determine how long the note should be played. The greater the value in $301, the longer the duration or length of the note. Memory location $302 is also used by the program, but the value stored in that location is set automatically by

Listing 7-2. Hexadecimal listing of Note Player.

```
303:A9 00 8D 02 03 AC 01 03 AE 00 03 AD
30 C0 CE 02 03 D0 03 88 F0 05 CA F0 EF
D0 F3 60
```

Listing 7-3. Note Tester.

```
10  REM   NOTE TESTER
20  POKE 768,160: POKE 769,160
30  CALL 771
40  POKE 769,32
50  CALL 771
60  POKE 768,32
70  CALL 771
80  POKE 769,160
90  CALL 771
```

the program. The program starts at byte 771 ($303). The statement CALL 771 causes the program to be executed. CALL 771 from BASIC is equivalent to the Monitor statement 303G.

A BASIC program can play a series of notes by repeatedly calling the Note Player program. The Random Notes program (Listing 7-4) selects random numbers for the pitch and duration of notes and places them in locations 768 and 769. Pressing CTRL-C causes the program to terminate.

Line 20: This line loads the Note Player program from the disk. Many machine language programs can only be used when they are placed in the particular memory segment for which they were developed. Note Player can be used from any memory location not reserved for some specific purpose by the computer. Since no address parameter is used, Note Player will be placed in memory at byte $303.

Lines 30-40: PLAY is assigned the address of the memory location (771) containing the first byte of the Note Player program. If an address parameter is used in line 20, the value of PLAY must be adjusted accordingly. The pitch of the note that is to be played must be stored in memory location 768. The pitch address is assigned to the variable PITCH in line 40.

Line 50: LNGTH is assigned the address of the location in memory that stores the duration of the note that is to be played. The letter E was omitted from the variable name since LEN is a reserved word in BASIC and cannot be used in a variable name.

Line 60: An ONERR branch to line 110 is set up in line 60. Since CTRL-C causes an error to occur, the program can be terminated by pressing CTRL-C. The program will continue to play randomly selected notes until it is interrupted by CTRL-RESET or CTRL-C.

Lines 70-80: A random number selected from the range 30 through 157 is generated and stored in the pitch memory location. Small numbers produce notes with high pitches that are mildly uncomfortable to the ear, so they have been omitted from the range of random pitches. Another random number, generated in line 80, is used as the duration of the note by poking it into the LNGTH memory byte.

Lines 90-100: Line 90 causes the Note Player program to play the randomly selected note, and the unconditional branch in line 100 results in an infinite loop that causes random notes to be played until the program is interrupted.

Listing 7-4. Random Notes.

```
10  REM   RANDOM NOTES
20  PRINT  CHR$ (4)"BLOAD NOTE PLAYER"
30 PLAY = 771
40 PITCH = 768
50 LNGTH = 769
60  ONERR  GOTO 110
70  POKE PITCH, RND (1) * 128 + 30
80  POKE LNGTH, RND (1) * 50 + 4
90  CALL PLAY
100  GOTO 70
110  END
```

Listing 7-5. Comb Sound.

```
10  REM       COMB SOUND
20  PRINT  CHR$ (4)"BLOAD NOTE PLAYER"
30 PLAY = 771
40 PITCH = 768:LNGTH = 769
50  POKE LNGTH,3
60 F = 10:L = 100
70  FOR N = F TO L
80  POKE PITCH,N: CALL PLAY
90  NEXT
100  END
```

Note Player can be used to create many different sound effects. The Comb Sound program (Listing7-5) produces a sound similar to that caused by raking the teeth of a comb across the edge of a tabletop.

The Comb Sound program plays all notes with pitches from 10 (F) through 100 (L) with each note held for a duration of 3. Different durations can be specified in line 50 to cause the program to produce various sound effects. The Random Comb Song (Listing 7-6) produces 10 distinct comb-raking sounds by randomly selecting the range of notes played. Each pass through the loop, which plays a range of notes, causes a different sound to be emitted.

Both Comb Sound and Random Comb Song play the highest pitched note in the selected range first. They can be modified to play the lowest pitched note in the range first by replacing the statement POKE PITCH,N with POKE PITCH,L+1–F.

Two Combs (Listing 7-7) is a program that alternates between the sounds made by playing the highest and lowest notes in the range. Random Comb Duet (Listing 7-8) combines the techniques used in Two Combs and Random Comb Song to produce its intriguing sounds.

MUSIC AND SONGS

Another application of the Note Player is creating music. Each musical note of the scale has a distinct pitch produced by a unique number of vibrations per second. For example, the A above middle C is the sound generated by 440 vibrations per second. Considering the high clock speed of the computer's processor (over a million vibrations per second), the speaker can be made to vibrate at a rate that will produce the sounds of musical notes.

Listing 7-6. Random Comb Song.

```
10  REM   RANDOM COMB SONG
20  PRINT  CHR$ (4)"BLOAD NOTE PLAYER"
30 PLAY = 771
40 PITCH = 768:LNGTH = 769
50  POKE LNGTH,3
55  FOR PASS = 1 TO 10
60 F =  RND (1) * 50 + 5:L =  RND (1) * 40 + 60
70  FOR N = F TO L
80  POKE PITCH,N: CALL PLAY
90  NEXT
100  NEXT PASS
```

Listing 7-7. Two Combs.

```
10  REM       TWO COMBS
20  PRINT  CHR$ (4)"BLOAD NOTE PLAYER"
30 PLAY = 771
40 PITCH = 768:LNGTH = 769
50  POKE LNGTH,3
60 F = 10:L = 100
70  FOR N = F TO L
80  POKE PITCH,N: CALL PLAY
85  POKE PITCH,L + 1 - N: CALL PLAY
90  NEXT
100  END
```

Listing 7-8. Random Comb Duet.

```
10  REM   RANDOM COMB DUET
20  PRINT  CHR$ (4)"BLOAD NOTE PLAYER"
30 PLAY = 771
40 PITCH = 768:LNGTH = 769
50  POKE LNGTH,3
55  FOR PASS = 1 TO 10
60 F =  RND (1) * 50 + 5:L =  RND (1) * 40 + 60
70  FOR N = F TO L
80  POKE PITCH,N: CALL PLAY
85  POKE PITCH,L + 1 - N: CALL PLAY
90  NEXT
100  NEXT PASS
```

By selecting the appropriate values for the pitch, Note Player can play the notes of the scale.

For example, if the value 81 is placed in $300 before executing the Note Player program, the sound produced is close to the A above middle C. Table 7-1 gives the pitch values necessary for Note Player to play all the notes of the scale in three different octaves. Since the pitch values listed are integer values, some of the sounds produced are not quite accurate. Those integer values which produce a frequency closest to the desired pitch are used. The Musical Scale program (Listing 7-9) uses Note Player to play all three octaves presented in Table 7-1.

With this vast selection of notes available, the notes of nearly any song can be played using the Note Player. Since the techniques used in the Musical Scale program can be utilized for playing songs on the computer, let's look at it line-by-line.

Table 7-1. Note Pitch Table.

Note	Low	Middle	High
F#	192	96	48
G	181	91	45
G#	171	85	43
A	161	81	40
A#	152	76	38
B	144	72	36
C	136	68	34
C#	128	64	32
D	121	60	30
D#	114	57	29
E	108	54	27
F	102	51	25

Listing 7-9. Musical Scale program.

```
10  REM      MUSICAL SCALE
20  PRINT  CHR$ (4)"BLOAD NOTE PLAYER"
30 PLAY = 771
40 PITCH = 768:LNGTH = 769
50  POKE LNGTH,50
60  READ NN
70  FOR N = 1 TO NN
80  READ SOUND
90  POKE PITCH,SOUND
100  CALL PLAY
110  NEXT N
120  END
130  DATA  36
140  DATA  192,181,171,161,152,144,136,128,121,114,108
     ,102
150  DATA  96,91,85,81,76,72,68,64,60,57,54,51
160  DATA  48,45,43,40,38,36,34,32,30,29,27,25
```

Lines 20-40: These lines load the Note Player program and initialize the variables used to reference the critical address needed to use it in a BASIC program.

Line 50: Each note played is held for a duration of 50. The value placed in the LNGTH address by line 50 may be increased if the notes need to be held for a longer period of time. Remember that the maximum value that can be placed in any byte of memory is 255.

Line 60: All the note pitches are stored in DATA statements. In line 60 the program reads a pitch value from a DATA statement, plays it, and then continues the process until all the notes have been played. NN is the variable used in Musical Scale to store the first value that appears in DATA. The first number in DATA is the Number of Notes the program is to play.

Lines 70-110: This loop plays each note pitch stored in DATA. The loop limit is set to NN, which is the Number of Notes to be played, in line 70. A DATA entry is READ in line 80 and assigned to the variable SOUND. Line 90 pokes the pitch value into byte 768 of memory. The Note Player program is executed in line 100, which causes a note of the scale to be played.

Lines 130-160: The number of notes the program is to play is stored in theDATA statement in line 130. Placing the number of notes in a data statement allows the program to play any number of notes by simply altering data; no other changes need to be made to the program. The pitch value of each note to be played is placed in the remaining DATA statements. One octave of the scale is in each DATA statement so that each can be distinguished easily.

The Fanfare Song program (Listing 7-10) is similar to the Musical Scale program discussed previously. Since the notes in Fanfare Song have varying durations, the statement that places the duration value into the LNGTH byte is located inside the loop that plays the notes of the song. Line 140 stores the number of notes in the song, but the remaining DATA statements are configured differently than in the Musical Scale program. Each note played by this program requires two DATA entries; the first value indicates the pitch, and the second value indicates the duration. The musical score from which the notes were taken is presented in Fig. 7-1.

Anyone who can read music should be able to place notes of songs into DATA statements for the Fanfare Song program to play; those who cannot should be able to use the program effectively by

Listing 7-10. Fanfare Song program listing.

```
10  REM       FANFARE SONG
20  PRINT  CHR$ (4)"BLOAD NOTE PLAYER"
30 PLAY = 771
40 PITCH = 768:LNGTH = 769
50  POKE LNGTH,50
60  READ NN
70  FOR N = 1 TO NN
80  READ SOUND,HOLD
90  POKE PITCH,SOUND: POKE LNGTH,HOLD
100   CALL PLAY
110   NEXT N
120   END
130   REM     SONG DATA
140   DATA 12
150   DATA 102,30,136,60,91,90
160   DATA 91,30,108,60,68,90
170   DATA 68,30,91,60,54,90
180   DATA 45,30,45,30,45,120
```

referring to Fig. 7-2, which illustrates the positions of the notes on the treble and bass staves.

The most common difficulty users have in coverting musical symbols to numeric notation is determining the correct durations for the notes. Some of the common note symbols and their relative durations are presented in Fig. 7-3. The correct duration value of a note is not easily determined, so experimentation is often required. Choose a number, say 80, to represent the duration of a quarter note. The duration of a half note is twice that of a quarter note, so 160 is used for the half note's duration. An eighth note would be 40, and so on.

Since songs vary in tempo, the duration values specified in Fig. 7-3 will not be suitable for every song. When selecting the duration for notes from a particular song, enter a few notes from the song and then run the program. If the tempo seems too quick, increase the quarter note duration and adjust the other notes accordingly. If the tempo is too slow, use a smaller number for the quarter note duration.

The Note Player machine language program has many uses. It also has one significant limitation.

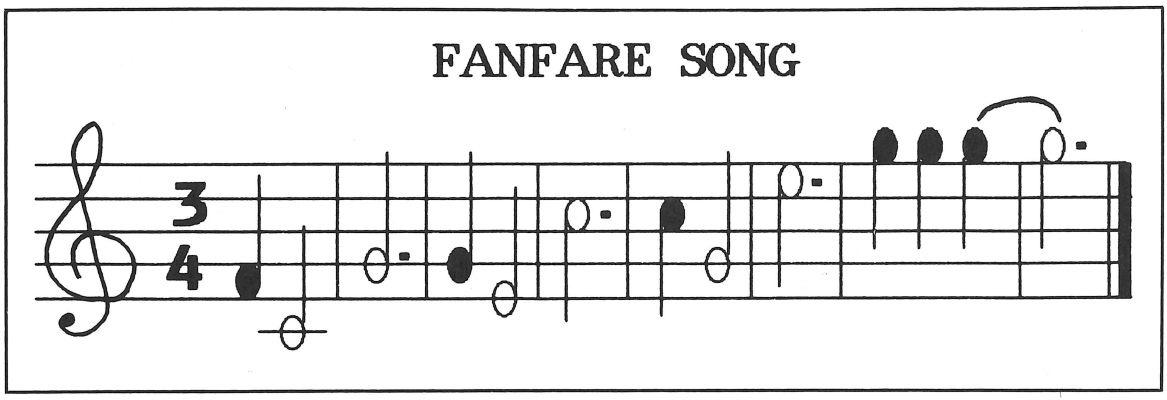


Fig. 7-1. This is the music score for the song played in the Fanfare Song program.

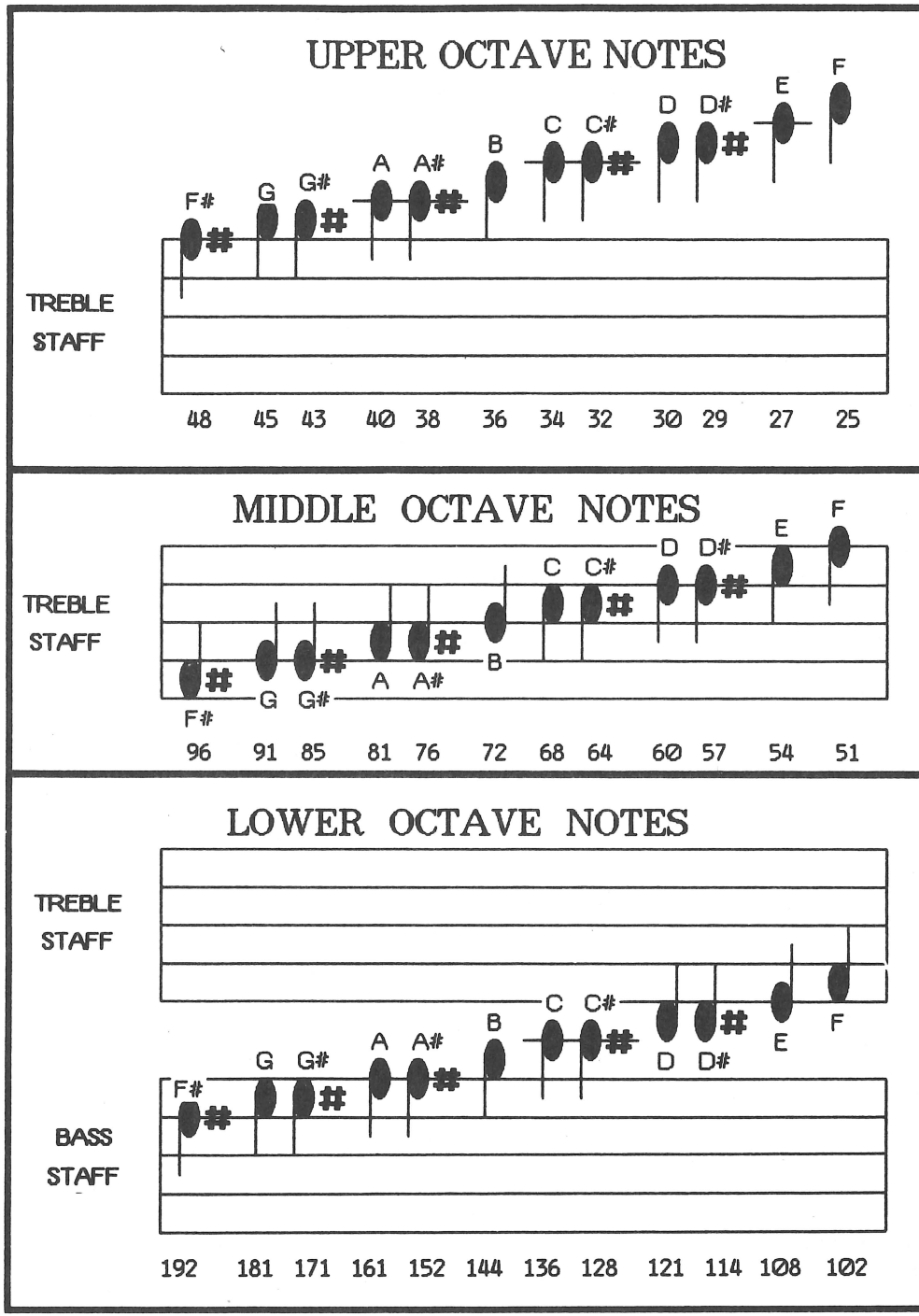


Fig. 7-2. Notes from three octaves can be played using the Note Player program. Pitches for the notes marked with sharps can be used for the flats of the notes to their right on the scale.

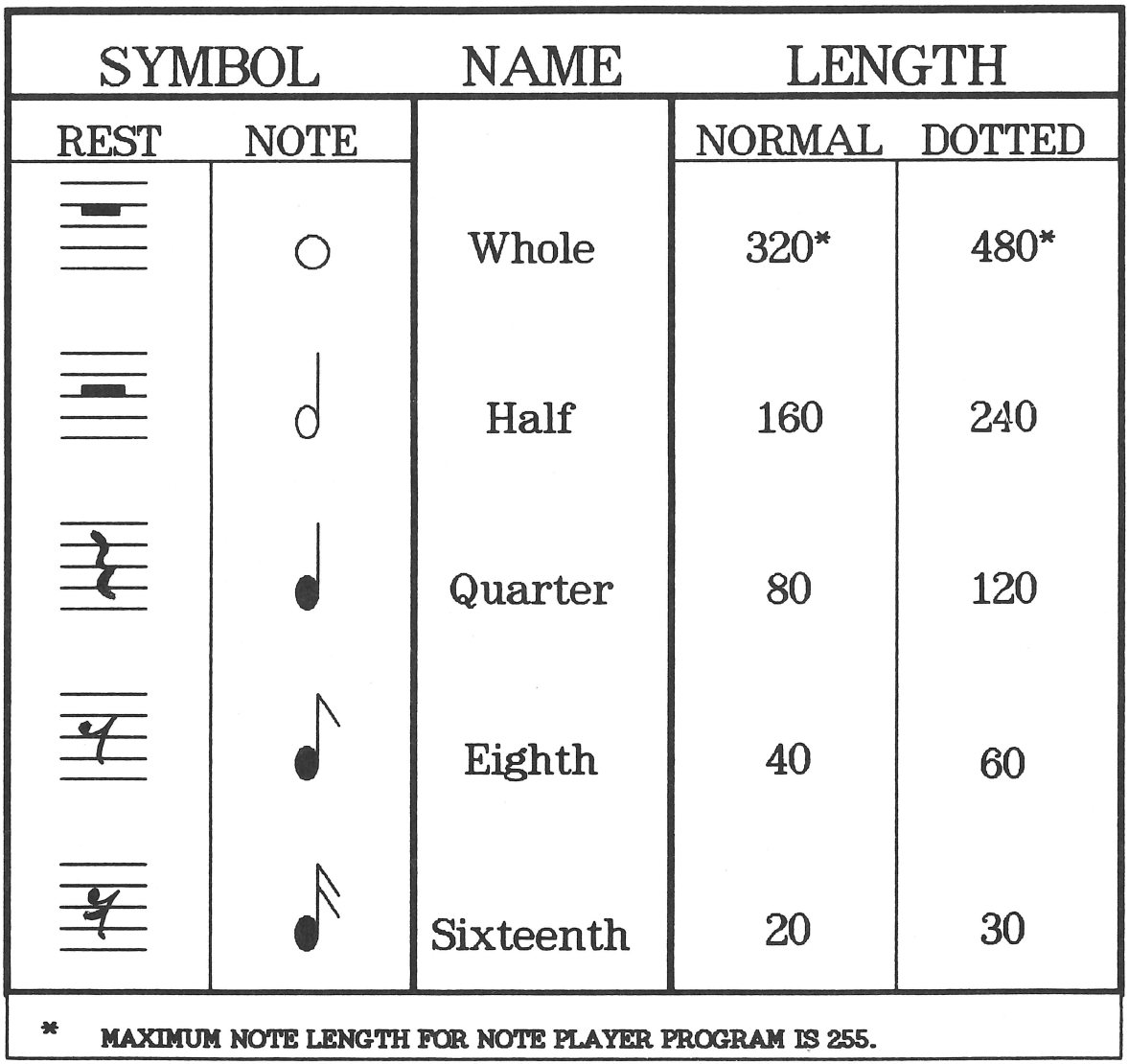


Fig. 7-3. Some of the common music symbols are shown in this figure. Their relative durations are given based on a quarter note length of 80.

Since only one byte of memory is used to store the duration of a note, the greatest duration value available is 255. Therefore, Note Player is not suitable for playing songs that have either a slow tempo or have notes that need to be held for longer periods of time.

The Musical Sounds machine language program (Listing 7-11) remedies the problem of note duration. It can hold notes having durations up to 65,535; it too has limitations. Unlike Note Player, which can be placed anywhere in RAM, Musical Sounds must be placed in the memory segment starting at $0306. It also requies a more complex data structure than the Note Player program.

The Musical Sounds hexadecimal listing (Listing 7-12) should be entered at $306 from the

Listing 7-11. Assembly listing of Musical Sounds program (continued on page 134).

```
1   *     MUSICAL SOUNDS
2
3             ORG   $306
4   SPKR      EQU   $C030
5   PITCH     EQU   $300
6   TIME1     EQU   $302
7   TIME2     EQU   $301
8   LSBCT     EQU   $304
9   MSBCT     EQU   $303
10  LBPTR     EQU   $0B
11  HBPTR     EQU   $0C
12
13
14  *  PLAY A NOTE FROM SONG
15
16  START     LDX   PITCH
17            BEQ   REST
18            LDA   SPKR
19  DELAY     DEY
20            BNE   PCNT
21            JMP   TIMER
22  PCNT      DEX
23            BEQ   START
24            JMP   DELAY
25  TIMER     DEC   TIME1
26            BNE   PCNT
27            LDA   TIME2
28            ORA   TIME1
29            BEQ   END
30            DEC   TIME2
31            JMP   PCNT
32
33  *  CHECK FOR END OF SONG
34
35  END       DEC   LSBCT
36            BNE   NEWNOTE
37            LDA   MSBCT
38            ORA   LSBCT
39            BEQ   QUIT
40            DEC   MSBCT
41            JMP NEWNOTE
42
43  *  RESET NOTE POINTER
44
45  QUIT      PLA
46            STA   HBPTR
47            PLA
48            STA   LBPTR
49
50  *  RETURN TO BASIC
51
52            RTS
53
54  *  PLAY A REST NOTE
55
56  REST      NOP
57            NOP
58            DEY
59            BNE   REST
60            SEC
61            LDA   TIME1
62            SBC   #$01
63            STA   TIME1
64            LDA   TIME2
65            SBC   #$00
66            STA   TIME2
67            ORA   TIME1
68            BEQ   END
69            JMP   REST
70
71  *  POINT TO START OF SONG
72
73  BEGIN     LDY   #$00
74            LDA   LBPTR
75            PHA
76            LDA   HBPTR
77            PHA
78            LDA   (LBPTR),Y
79            STA   MSBCT
80            INY
81            LDA   (LBPTR),Y
82            STA   LSBCT
83            CLC
84            INC   LBPTR
85            INC   LBPTR
86            BCC   NEWNOTE
87            INC   HBPTR
88
89  *  GET NEXT NOTE FROM SONG
90
91  NEWNOTE LDY   #$00
92  SETNOTE LDA   (LBPTR),Y
93            STA   PITCH,Y
94            INY
95            CPY   #$03
96            BNE   SETNOTE
97
98  *  FIND NOTE PITCH IN TABLE
```

```
99
100          LDX  PITCH
101          LDA  DATA,X
102          STA  PITCH
103
104  *  ADVANCE NOTE POINTER
105
106          CLC
107          LDA  LBPTR
108          ADC  #$03
109          STA  LBPTR
110          LDA  HBPTR
111          ADC  #$00
112          STA  HBPTR
113          JMP  START
114
115  *  NOTE PITCH TABLE
116
117 DATA   DFB  0,192,181,171,
                161,152,144,
                136,128,121,
                114,108,102
118          DFB  96,91,85,81,76,
                72,68,64,60,57,
                54,51
119          DFB  48,45,43,40,38,
                36,34,32,30,29,
                27,25
```

Monitor and saved to the disk using the statement

BSAVE MUSICAL SOUNDS, A$306, L197

Musical Sounds requires a binary data file containing the notes that are to be played. The first two bytes of the song data file indicate the number of notes in the song, expressed using three 2-digit hex codes. The first code is a number from 0 through 36 representing the pitch of the note, and the other two numbers tell the program how long to hold the note. The program can play all 36 notes presented earlier in this chapter. A song can theoretically consist of up to 65,535 notes, but the limited memory of the computer argues against using more than 500 notes in each song file.

Due to the complex note data organization required by the Musical Sounds program, a special program called Song Editor (Listing 7-13) was developed. The program builds song data files for songs which the Musical Sounds program can play. Musical Sounds must be on the disk containing the Song Editor.

A detailed discussion of the Song Editor is

Listing 7-12. Hex listing of Musical Sounds.

```
306:AE 00 03 F0 3C AD 30 C0 88 D0 03 4C
1A 03 CA F0 EF 4C 0E 03 CE 02 03 D0 F5
AD 01 03 0D 02 03 F0 06 CE 01 03 4C 14
03 CE 04 03 D0 4F AD 03 03 0D 04 03 F0
06 CE 03 03 4C 81 03

:68 85 0C 68 85 0B 60 EA EA 88 D0 FB 38
AD 02 03 E9 01 8D 02 03 AD 01 03 E9 00
8D 01 03 0D 02 03 F0 CB 4C 47 03 A0 00
A5 0B 48 A5 0C 48 B1 0B 8D 03 03 C8 B1
0B 8D 04 03 18 E6 0B

:E6 0B 90 02 E6 0C A0 00 B1 0B 99 00 03
C8 C0 03 D0 F6 AE 00 03 BD A6 03 8D 00
03 18 A5 0B 69 03 85 0B A5 0C 69 00 85
0C 4C 06 03 00 C0 B5 AB A1 98 90 88 80
79 72 6C 66 60 5B

:55 51 4C 48 44 40 3C 39 36 33 30 2D 2B
28 26 24 22 20 1E 1D 1B 19
```

Listing 7-13. Song Editor (continues on page 136).

```
5  REM  SONG EDITOR
6  REM
10  HIMEM: 36863
20  PRINT  CHR$ (4)"BLOAD MUSICAL SOUNDS"
30  DEF  FN MOD(Q) =  INT ((Q / 256 -  INT (Q / 256)) *
     256)
40  DEF  FN DIV(Q) =  INT (Q / 256)
50 SD = 36864
60  DIM PC$(36)
70  FOR X = 0 TO 36: READ PC$(X): NEXT
80  POKE SD,0: POKE SD + 1,0
90  HOME : INPUT "NAME THE SONG FILE:";SF$
100  ONERR  GOTO 120
110  PRINT  CHR$ (4)"BLOAD"SF$
120  POKE 216,0
130  HOME : VTAB 24: PRINT "<Q>UIT  <P>LAY  <U>P  <I>N
     SERT  <Z>AP";
140 NN =  PEEK (SD) * 256 +  PEEK (SD + 1)
150 N = 0
160  POKE 35,22: HOME : PRINT " #      PITCH      DURAT
     ION": PRINT " =     =====      ========": POKE 34
     ,3: HOME
165  REM ----------------------
166  REM   NOTE ENTRY MODULE
167  REM ----------------------
170  PRINT :Z =  PEEK (37): IF N + 1 > NN THEN 200
180 OP$ = PC$( PEEK (SD + 2 + 3 * N)):OD =  PEEK (SD +
     4 + 3 * N) + 256 *  PEEK (SD + 3 + 3 * N)
190 P$ = OP$:D$ =  STR$ (OD)
200  VTAB Z: PRINT N + 1".";: HTAB 10: PRINT P$;: HTAB
     10: INPUT "";P$: IF  NOT  LEN (P$) THEN P$ = OP$:
      VTAB Z: HTAB 10: PRINT P$
210  IF P$ = "Z" AND N > 0 THEN 500
220  IF P$ = "I" THEN 1000
230  IF P$ = "Q" THEN 2000
240  IF P$ = "P" THEN 1500
250  IF P$ = "U" THEN N = N - 1 * (N > 0): VTAB Z: CALL
      - 958: VTAB Z - 1 * (Z > 4): GOTO 170
260  FOR X = 0 TO 36: IF P$ = PC$(X) THEN 290
270  NEXT X: PRINT  CHR$ (7)
280 P$ = OP$: VTAB Z: CALL  - 868: GOTO 170
290  IF N > 500 THEN 280
300 N = N + 1: POKE SD - 1 + 3 * N,X: IF N > NN THEN N
     N = N
310  VTAB Z: HTAB 20: PRINT D$;: HTAB 20: INPUT "";D$
320 Q =  VAL (D$): IF  NOT  LEN (D$) THEN Q = OD: VTAB
     Z: HTAB 20: PRINT Q
330  IF Q < 1 OR Q > 65535 THEN  VTAB Z: HTAB 120: PRINT
      CHR$ (7);: CALL  - 868: GOTO 310
```

```
340  POKE SD + 3 * N, FN DIV(Q)
350  POKE SD + 1 + 3 * N, FN MOD(Q)
360 P$ = "":D$ = "":OP$ = "":OD = 0: GOTO 170
488  REM ----------------------
489  REM     INSERT A NOTE
490  REM ----------------------
500  FOR X = SD + 2 + 3 * N TO NN * 3 - 1 + SD
510  POKE X, PEEK (X + 3)
520  NEXT
530 NN = NN - 1: IF N = NN THEN N = N - 1 * (N > 0): VTAB
     Z: CALL  - 958: VTAB Z - 1 * (Z > 4): GOTO 360
540  GOTO 180
990  REM ----------------------
991  REM     DELETE A NOTE
992  REM ----------------------
1000  FOR X = NN * 3 + 5 + SD TO SD + 5 + 3 * N STEP  -
     1
1010  POKE X, PEEK (X - 3): NEXT
1020 NN = NN + 1
1030 OP$ = "":OD = 0: GOTO 190
1490  REM ----------------------
1491  REM      PLAY THE SONG
1492  REM ----------------------
1500 Q = N: IF  NOT Q THEN 1540
1510  POKE 11,0: POKE 12,144
1520  POKE SD, FN DIV(Q): POKE SD + 1, FN MOD(Q)
1530  CALL 869
1540 P$ = OP$
1550  GOTO 280
1990  REM ----------------------
1991  REM      SAVE THE SONG
1992  REM ----------------------
2000 Q = NN: IF  NOT Q THEN 2050
2010  POKE SD, FN DIV(Q): POKE SD + 1, FN MOD(Q)
2020  TEXT : HOME : PRINT "NAME THE SONG:"SF$;: HTAB 1
     5: INPUT "";S$: IF  LEN (S$) THEN SF$ = S$
2030  VTAB  PEEK (37): HTAB 15: PRINT SF$
2040  PRINT  CHR$ (4)"BSAVE"SF$",A$9000,L"Q * 3 + 3
2050  END
2490  REM ----------------------
2491  REM       NOTE NAMES
2492  REM ----------------------
2500  DATA R,F#-,G-,G#-,A-,A#-,B-,C-,C#-,D-,D#-,E-,F-
2510  DATA F#,G,G#,A,A#,B,C,C#,D,D#,E,F
2520  DATA F#+,G+,G#+,A+,A#+,B+,C+,C#+,D+,D#+,E+,F+
```

provided in order to demonstrate how to use the program to create song files and how to use the machine language program Musical Sounds from within a BASIC program. Read the discussion before trying to use either program.

Lines 10-20: High Memory is set to 36863, which is $8FFF. Data for songs will be placed in memory from $9000 to $95FF. HIMEM must be set

to keep the song from being clobbered by BASIC strings used in the program. Line 20 loads Musical Sounds into memory at $306.

Lines 30-40: Applesoft BASIC does not have a MOD or DIV function, so the functions must be defined using the DEF FN statement whenever they are needed. The MOD function defined in line 30 finds the *remainder* of a number Q when it is divided by 256, while the DIV function finds the integer quotient of a number Q when it is divided by 256. For example, if 300 is divided by 256, the answer is 1 remainder 44. The MOD function gives a result of 44, and the DIV function returns a value of 1. These functions are needed so that numbers greater than 255 (the greatest value that can be placed in one byte of memory) can be used in the song data file. Numbers greater than 255 must be placed in two memory bytes. The MOD function determines the value that is to be placed in the second or *low* byte, and the DIV function determines the value that is to be placed in the first or *high* byte.

Line 50: SD is the variable that stores the starting memory address, $9000, for the song data file.

Lines 60-70: Musical Sounds can play 36 different pitches spanning three octaves. Each octave consists of 12 distinct pitches: F#, G, G#, A, A#, B, C, C#, D, D#, E and F. These 12 notes appear on the treble staff. Notes one octave higher are denoted with a plus sign (+), and those one octave lower are denoted with a minus sign (–). The array PC$, used to store the pitch codes, is dimensioned in line 60. The pitch codes are placed in the array from data in line 70. The code R is used to denote a rest. Figure 7-4 shows the position of each note with respect to the treble cleff.

Line 80: Song Editor allows song data files stored on the disk to be edited. In order to ensure that the program does not get mixed up when new programs are being edited, line 80 places the number 0 in the first two bytes of the song data memory segment to inform the program that no notes have been entered yet.

Line 90: After clearing the text screen, the program asks for the name of the song data file that is to be edited. The name of the file is assigned to the variable SF$.

Lines 100-120: These lines are an error routine. If the filename specified in line 90 is not on the disk in the drive, the program assumes that the user wants to start a new file.

Line 130: This line displays prompts indicating the main options of the program. The program only recognizes prompt instructions when asking for a note pitch.

Line 140: The variable NN, used to stored the number of notes in the song, is initialized in line 140. If a song file has been loaded, the number of notes in the song will be in the first two bytes of the song data memory segment. If no song has been loaded its value will be 0, since that value was placed in memory before the attempt to load a file.

Line 150: The variable N keeps track of which note of the song is being entered. It is initialized to zero so that the editing process will start with the first note of the song.

Line 160: The headers #, Pitch, and Duration are placed at the top of the screen. When the cursor is under Pitch, enter a pitch code or an editor instruction. When the cursor is under Duration, enter a number from 1 to 65,535 indicating how long the Musical Sounds program is to hold the pitch. The relative position in the song of the note being entered is displayed under the # header. The headers and option prompts will not be scrolled from the screen because the text window is lowered to the 4th line of the screen by the statement POKE 34,3 and the bottom of the window is set to the 23rd screen row by the statement POKE 35,22.

Lines 170-190. The note entry module begins in line 170. Z is assigned the current vertical position of the cursor. This value is needed to ensure that the cursor is properly placed on the screen during editing. Line 170's primary purpose is to determine if a note exists in the Nth position of the song. If not, the program branches to line 200 to obtain a note or instruction.

If a note is already in the Nth position, it is obtained from the song data segment and assigned to the variable OP$ (meaning Old Pitch), in line 180. The note's duration is assigned to the variable

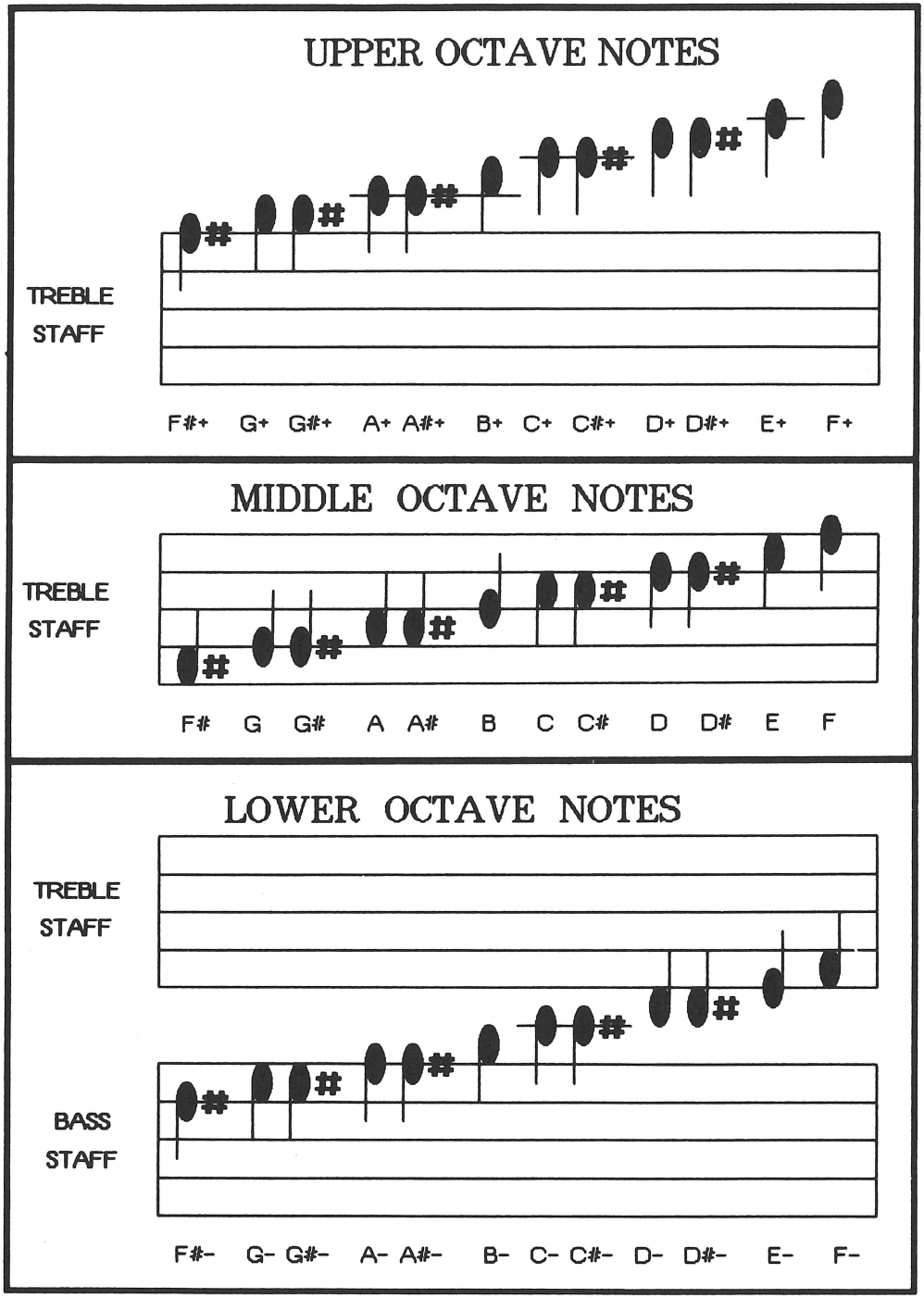


Fig. 7-4. The Music Editor program allows the notes to be represented by codes that are similar to their proper names.

OD (meaning Old Duration). P$ is the variable used to store the note entered from the keyboard. It is assigned the old pitch code in line 190. D$ is the variable used to indicate the duration of the note being entered. It is initialized to the old duration value in line 190.

Line 200. This line displays the old pitch code and old duration values for the note being entered, if any. The new pitch code is then input and assigned to the variable P$. If no entry is made before pressing Return, the old pitch value displayed on the screen is retained in the song and the code is reprinted on the screen.

Line 210. Editing instructions are accepted when the program requests a pitch code. Line 210 checks to see if the Z (Zap) instruction was entered. The Zap instruction removes the note in the current position from the song. The program statements which remove a note start at line 500. This branch will not be taken if no notes have been entered.

Line 220. When the I (Insert) instruction is received, line 220 sends the program to line 1000, where room for one note will be cleared.

Line 230. The Quit option is used when all the notes of the song have been entered or edited. When Q is entered for the pitch, the program branches to line 2000 to save the file on disk.

Line 240. Entering P for Play causes the first N notes of the song to be played. If the entire song is to be played, the cursor must be positioned beyond the last note in the song. Line 240 calls the program lines that cause the song to be played; these are located starting at line 1500.

Line 250. U is used to tell the computer to back up one note in the song. This allows the user to correct notes after they have been entered. All notes removed from the screen when the U option is selected remain in the song, although they can no longer be seen. They will reappear on the screen when the editor returns to the notes in those positions.

Lines 260-280. Line 260 checks to see if the pitch code entered for the Nth note is known by the program. If it is found, the program branches to line 290 to find out the duration of the note. If the code is illegal, the program beeps, reinstates the old pitch value, and branches to line 170 to obtain a new pitch for the note.

Line 290: The maximum number of notes that can be entered for one song is 500. Line 290 enforces this restriction.

Line 300. The note position pointer N is updated in line 300. In addition, the subscript of the element in the PC$ array is placed in the song data segment. Line 300 also ensures that the variable which keeps track of the number of notes in the song is accurate.

Line 310. The program obtains the duration for the Nth note of the song in line 310, and assigns it to D$.

Lines 320-330. The duration input is checked for accuracy in these lines. The value of the entry is assigned to the variable Q. If no duration was entered, the old duration value (if any) if used. If Q is not within the legal range for note durations (1-65535) the duration must be entered again.

Lines 340-350. Lines 340 and 350 place the duration for the Nth note in the appropriate location in the song data segment. The DIV and MOD functions are used to convert the duration to its two byte form.

Line 360. All the variables used in obtaining the note from the keyboard are reinitialized in line 360, and the program branches to line 170 to get the next note in the song.

Lines 500-540. Notes can be deleted from the song by moving all the notes after the current note position forward one note position. Since each note requires three bytes, each byte in the song data segment must be moved three bytes forward. This "slide" is accomplished in lines 500 to 520. The number of notes in the song is reduced by one in line 530; then the program branches back to line 180 to get the next note of the song.

Lines 1000-1030. In order to insert a note in the song, all the song data from the current note position to the end of the song must be moved three bytes toward the end of the song data segment to make room for the new note. After the slide is performed by lines 1000-1010, the number of notes in the song is modified in line 1020. The program then returns to the note entry module.

Lines 1500-1540. These lines are very important because they demonstrate how to use the Musical Sounds machine language program from within a BASIC program. Particular attention should be given to lines 1520 and 1530.

The number of notes that are to be played is assigned to the variable Q in line 1500. Line 1510 places the value of Q in the first two bytes of the song data segment. Musical Sounds needs to know where the song data is stored in memory, expecting to find the starting address in bytes 11 and 12. The address is calculated using the technique described earlier for giving the starting address of a shape table. The song data segment in this program starts at $9000. The decimal equivalent of the first two digits, 90, is 144, and the decimal equivalent of the second two digits of the address is 0. The decimal value of the second pair of digits must be placed in memory location 11, and the decimal value of the first pair must be placed in location 12. These values must be poked into the appropriate memory locations before the Musical Sounds program is called.

The Musical Sounds program physically starts at memory location $306, but the logical beginning is at $365. This means that the execution of the program must begin at $365. The decimal equivalent of $365 is 869, so the statement CALL 869 instructs the computer to play the song. Executing a machine language program from a BASIC is much like executing a BASIC subroutine. When the machine language program is finished, the BASIC program will resume execution starting with the statement immediately following the CALL command. In this case, when the machine language program is finished, line 1540, which causes a branch to line 280, will be executed.

Lines 2000-2050. These program lines save the song data segment on disk. Lines 2000 and 2010 place the number of notes in the song into the first two bytes of the song data segment if necessary. Line 2020 obtains the filename that is to be used for storing the song data on the disk. If the user merely presses the Return key, the filename specified at the beginning of the program will be used. The data is saved to the disk in line 2040; then the program ends.

Lines 2500-2510. The legal pitch codes appear in these DATA statements. Thirty-six distinct pitches are available. Those of you who have solid musical backgrounds may find some of the pitches to be a little flat. It is also possible that the pitch codes do not exactly correlate to the proper musical pitch. Melodies entered using these codes will be fairly accurate, but may seem to be in a different key than the original piece. Those who would like to try their hands at developing a more accurate set of pitches are encouraged to try. The numbers that determine the pitches used by the Musical Sounds program are located in lines 117 through 119 of the assembly listing. Changes in the pitch codes used in the Song Editor program do not change the pitches played by the Musical Sounds program.

ARCADE GAME SOUNDS

Many sound effects used in games cannot be classified as music, and therefore cannot easily be

Listing 7-14. Laser Down assembly listing.

```
1                       ORG   $304
2     FIRST             EQU   $300
3     LAST              EQU   $301
4     NOTE              EQU   $302
5     SPEAKER           EQU   $C030
6
7     *  LASER DOWN  *
8
9                       LDY   #50
10    START             LDA   #$30
11                      STA   FIRST
12                      LDA   #$A0
13                      STA   LAST
14    PITCH             LDX   FIRST
15                      STX   NOTE
16                      LDA   SPEAKER
17    DELAY             DEC   NOTE
18                      BNE   DELAY
19                      INC   FIRST
20                      LDA   FIRST
21                      CMP   LAST
22                      BNE   PITCH
23                      DEY
24                      BNE   START
25                      RTS
```

Listing 7-15. Laser Up assembly listing.

```
1                       ORG   $304
2     FIRST             EQU   $300
3     LAST              EQU   $301
4     NOTE              EQU   $302
5     SPEAKER           EQU   $C030
6
7     *   LASER UP   *
8
9                       LDY   #50
10    START             LDA   #$30
11                      STA   LAST
12                      LDA   #$A0
13                      STA   FIRST
14    PITCH             LDX   FIRST
15                      STX   NOTE
16                      LDA   SPEAKER
17    DELAY             DEC   NOTE
18                      BNE   DELAY
19                      DEC   FIRST
20                      LDA   FIRST
21                      CMP   LAST
22                      BNE   PITCH
23                      DEY
24                      BNE   START
25                      RTS
```

generated with the Musical Sounds or Note Player programs. Instead, a single, simple machine language program can be used to generate most of the game sounds desired. Two short machine language programs, Laser Down and Laser Up, appear in Listings 7-14 and 7-15. With very minor modifications they can generate sounds ranging from a car that won't start to a telephone ringing to a space weapon.

The hexadecimal listing of Laser Down (Listing 7-16) should be entered starting at memory location $304, and saved on the disk using the statement

BSAVE LASER DOWN, A$304, L41.

The hexadecimal listing (Listing 7-17) of the Laser Up program can be entered and saved by following the instructions given for the hexadecimal listing of the Laser Down program.

The Laser Up and Laser Down programs generate very similar sounds. Both play sequentially all the notes in the range specified by bytes $307 (the highest note) and $30C (the lowest note). Laser Up plays the lowest note of the range first, while Laser Down plays the highest first. In both programs the notes are played very quickly, so the effect is like blowing a slide trombone while moving the slide. The sliding sound is more pronounced as the interval between the notes is increased. The values stored in byte $305 tell the computer how many times the sound is to be repeated; varying the contents of this byte varies the length of time the sound is produced.

Random Laser (Listing 7-18) is a program that

Listing 7-16. Hexadecimal listing of Laser Down.

```
304:A0 32 A9 30 8D 00 03 A9 A0 8D 01 03
AE 00 03 8E 02 03 AD 30 C0 CE 02 03 D0
FB EE 00 03 AD 00 03 CD 01 03 D0 E7 88
D0 DA 60
```

Listing 7-17. Hexadecimal listing of Laser Up.

```
304:A0 32 A9 30 8D 01 03 A9 A0 8D 00 03
AE 00 03 8E 02 03 AD 30 C0 CE 02 03 D0
FB CE 00 03 AD 00 03 CD 01 03 D0 E7 88
D0 DA 60
```

Listing 7-18. Random Laser.

```
10  REM     RANDOM LASER
20 FI = 775::LA = 780:RP = 773
30  PRINT  CHR$ (4)"BLOAD LASER UP"
40  HOME : PRINT "FIRST           LAST": PRINT "=====
          ====": POKE 34,3: HOME
50 FS =  RND (1) * 256
60 LS =  RND (1) * (256 - FS) + FS
70  POKE RP,20
80  POKE FI,FS: POKE LA,LS
90  PRINT  INT (FS), INT (LS).
100  CALL 772: GET G$: IF G$ <  > "Q" THEN 50
110  TEXT : END
```

can use either machine language laser sound program. It randomly selects values for $307 and $30C. Each interval is played 20 times. The first and last notes of the interval are printed in columns on the screen before the sound is played. After the machine language program has been executed, another sound can be generated by simply pressing any key except Q. Pressing Q causes the program to end.

We suggest that you jot down the first and last note values that generate usable sounds. Laser Down can be used with the program by changing line 20. The number in line 70 can be changed if repetitions rather than 20 are desired.

The statements in Lines 20, 30, 70, 80, and 100 of the Random Laser program must be included in any BASIC program that is to use the machine language laser sound programs.

This chapter has dealt primarily with sound applications that are not directly related to graphics programming. Graphics applications of the machine language sound-generating programs presented in this chapter are discussed and demonstrated in Chapter 8.

8

Combining High-Resolution Graphics Techniques

Many of the programming techniques discussed in Chapters 3 through 7 can be combined to produce high-quality programs. Several programs are presented in this chapter that demonstrate how to utilize all the high-resolution graphics and sound capabilities of the computer. We will discuss vector lines and designs, plot graphics enhancements of vector graphics, and utilizing polar equations, shapes and sounds.

VECTOR LINES AND DESIGNS

A practical shape definition which can be included in a shape table is that of a single vector departing a shaded grid. Such a shape can be used for many applications. When displayed at SCALE=1, it is a single point; at greater SCALE values it is a line. Programs that demonstrate some of the ways in which a single vector shape can be used are discussed in this section. The One Vector shape table listed below should be entered and saved on disk before reading the remainder of this section. The shape table is small enough to be placed in memory starting at $300. One Vector can be saved using the following statement:

```
BSAVE ONE VECTOR, A$300, L$6.
```

Here is the One Vector shape table:

```
300:01 00 04 00 05 00
```

Vector graphics can be used to create displays similar to those previously accomplished using high-resolution plot graphics. Vector Circle (Listing 8-1) is a program that uses the One Vector shape table to create a circular design.

The One Vector shape table is very short (6 bytes), so it can be placed in memory at $300. Line 30 places the starting address of the shape table in locations 232 and 233.

Line 40: Page 1 of high-resolution graphics is used for displaying the circular design. Line 40 specifies the color and sets the SCALE value, which determines the radius of the circle. The interior of the circle displayed is not solid. Since

Listing 8-1. Vector Circle.

```
10  REM      VECTOR CIRCLE
15  HOME
20  PRINT  CHR$ (4)"BLOAD ONE VECTOR,A$300"
30  POKE 232,0: POKE 233,3
40  HGR : SCALE= 40: HCOLOR= 3
60  FOR RO = 0 TO 64
70  ROT= RO
80  DRAW 1 AT 140,70
100  NEXT RO
105  VTAB 22: PRINT "PRESS ANY KEY.";: GET G$
110  HGR : TEXT : HOME
```

only 64 rotation positions can be used for displaying shapes, any circle with a circumference greater than 64 will not be solid in color. If solid circles are desired, the greatest radius, or SCALE, that can be used is 20.

Line 60: The circular design is formed by drawing the single vector shape at the center of the circle. The vector shape is then rotated 1/64th of a revolution and drawn again. Line 60 is the beginning of a loop that repeats the process 64 times to complete the design. RO is the variable which stores the ROtation value for drawing the radii of the circle.

Lines 70-80: Line 70 sets the ROT value for displaying the single vector shape. Each time the single vector shape is drawn, another line from the center to the exterior of the circle is displayed by line 80. All the lines formed by the single vector shape originate at the center of the circle. The circle can be placed at other areas of the screen by changing the AT parameters of the DRAW statement.

Lines 105-110: The completed circle is left on the screen until a key is pressed. At that time, the high-resolution screen is cleared, and the text page is displayed.

All the power of shape table-related BASIC commands can be applied to a single vector shape as easily as to the most complex shape. The Vector Circle program utilized a constant SCALE value while changing the ROT value to create its display. The Vector Spiral program (Listing 8-2) allows both the ROT and SCALE values to vary each time the shape is displayed. The resulting display is a spiral design.

The Vector Spiral program is similar to the Vector Circle. As mentioned above, the SCALE value is increased each time the shape is displayed.

Listing 8-2. Vector Spiral.

```
10  REM      VECTOR SPIRAL
15  HOME
20  PRINT  CHR$ (4)"BLOAD ONE VECTOR,A$300"
30  POKE 232,0: POKE 233,3
40  HGR
60  FOR RO = 0 TO 64
70  ROT= RO: SCALE= RO + 1
80  XDRAW 1 AT 140,70
100  NEXT RO
105  VTAB 22: PRINT "PRESS ANY KEY.";: GET G$
110  HGR : TEXT : HOME
```

Listing 8-3. Vector Spin.

```
10  REM     VECTOR SPIN
15  HOME
20  PRINT  CHR$ (4)"BLOAD ONE VECTOR,A$300"
30  POKE 232,0: POKE 233,3
40  HGR : SCALE= 40
50  FOR REV = 1 TO 5
60  FOR RO = 0 TO 64
70  ROT= RO
80  XDRAW 1 AT 140,70
90  XDRAW 1 AT 140,70
100  NEXT RO,RE
110  HGR : TEXT : HOME
```

The SCALE value is restricted to positive numbers in the program. If it is allowed to be 0 the resulting line is 256 screen dots long, which distracts from the appearance of the spiral.

No HCOLOR statement appears in the Vector Spiral program. XDRAW is used to display the shape, and XDRAW does not require the HCOLOR value to be specified. It checks the color of each dot on the screen used for displaying the shape, and simply changes them to the color's complement. Since all the dots are black before the shape is drawn, they are changed to white and the XDRAW command in line 80 is executed.

The Vector Spin program (Listing 8-3) demonstrates a technique for creating rotary motion. The program is very similar to the Vector Circle program presented earlier. However, after a scaled vector is placed on the screen, it is removed before being placed in the next rotation position.

Line 50 of the Vector Spin starts a loop that allows the spinning action to continue for five revolutions. The loop is terminated in line 100.

No HCOLOR statement appears in the program. Line 80 causes the shape to be displayed with white because the dots used were formerly displayed in black. Line 90 causes the dots used to display the shape to be changed back to the color black. Recall that the XDRAW command only works properly when the shape definitions have just one plotting vector departing from each shaded grid in the design.

The Vector Spiral Spin program (Listing 8-4) demonstrates that rotary motion techniques can be applied to the Vector Spiral program as easily as to

Listing 8-4. Vector Spiral Spin.

```
10  REM   VECTOR SPIRAL SPIN
15  HOME
20  PRINT  CHR$ (4)"BLOAD ONE VECTOR,A$300"
30  POKE 232,0: POKE 233,3
40  HGR
50  FOR REV = 1 TO 5
60  FOR RO = 0 TO 64
70  ROT= RO: SCALE= RO + 1
80  XDRAW 1 AT 140,70
90  XDRAW 1 AT 140,70
100  NEXT RO,RE
110  HGR : TEXT : HOME
```

Listing 8-5. Vector Donut.

```
10  REM      VECTOR DONUT
15  HOME
20  PRINT  CHR$ (4)"BLOAD ONE VECTOR,A$300"
30  POKE 232,0: POKE 233,3
40  HGR : SCALE= 20: HCOLOR= 0
50  FOR REV = 1 TO 5
60  FOR RO = 0 TO 64
70  ROT= RO
75  DRAW 1 AT 140,70
80  XDRAW 1
100  NEXT RO,RE
110  HGR : TEXT : HOME
```

the Vector Circle program.

Scaled single-vector shapes can also be used to place other shapes on the screen to form patterns. After the DRAW command has been used to plot a shape, another DRAW command (without the AT parameter) causes the specified shape to be drawn. This shape will start at the last point of the previous shape. The Vector Donut program (Listing 8-5) demonstrates this technique by drawing a scaled vector using black. The scaled vector shape is then placed on the screen in white using the XDRAW command. The process is repeated for all possible values of ROT to form a donut design.

Scaled vectors can also be used for creating displays of polygons. The SCALE value determines the length of the sides of the polygons. Vector Polygons (Listing 8-6) displays polygons having from 3 to 16 sides. The more sides the polygon has, the larger it appears on the screen; This is because the lengths of the sides do not vary as they did in the high-resolution graphics version of the program presented in Chapter 3.

Using shapes to create the illusion of motion was discussed in Chapter 5. Another type of motion, not previously discussed, is *spinning*. Spinning motion is accomplished by displaying a shape, erasing it, changing the ROT value, then repeating the display cycle. A problem arises when this technique is used, however. When shapes are displayed at angles that are not integral multiples of 90

Listing 8-6. Vector Polygons.

```
10  REM   VECTOR POLYGONS
15  HOME
20  PRINT  CHR$ (4)"BLOAD ONE VECTOR,A$300"
30  POKE 232,0: POKE 233,3
35  HGR : HCOLOR= 3: SCALE= 20
36  HPLOT 100,10
40  FOR PO = 3 TO 16
45  VTAB 23: PRINT PO" SIDES."
50  FOR RO = 0 TO 63 STEP 64 / PO
55  ROT= RO
60  XDRAW 1
70  NEXT RO
80  VTAB 21: PRINT "PRESS ANY KEY.";: GET G$
90  HGR : HOME : NEXT PO
100  TEXT : END
```

degrees, they increase in size. In order to avoid this problem, a shape definition designed specifically for other desired angles must be included in the table. Acceptable spinning motion can generally be achieved if the shapes that are to be used in spin sequences are defined at a 90-degree orientation and a 45-degree orientation.

The Arrows shape table provided below contains two distinct definitions for an arrow (See Fig. 8-1). The first arrow shape can be displayed using right-angle rotation values such as 0, 16, 32, and 48. The second arrow shape can be displayed using 45-degree rotations values such as 8, 24, 40, and 56. A single-vector shape definition is included in the table as shape number 3.

The programs presented in this chapter that demonstrate spinning motion require the use of the Arrow shape table. Enter the Arrow shape table starting at $6000. The table can be saved using the statement

```
BSAVE ARROW, A$6000, L36.

6000:03 00 08 00 15 00 22 00 24 24 3C 3F
     OC OC OC 15 15 15 3F 07 00 07 38 38 38
     60 05 38 3F 37 36 2E 20 00 04 00
```

The Arrow Spin I program (Listing 8-7) demonstrates the technique of using two similar shapes to create a spinning motion. The disk containing the Arrow shape table must be in the drive before running the program.

Lines 15-40 perform the preliminary tasks necessary for displaying designs from the Arrow shape table on page 1 of high-resolution graphics.

In lines 50-70, REV is the variable used to control the number of REVolutions (10) made by the spinning action. RO is the variable which determines the rotation value to be used for displaying the arrow shapes on the screen. the loop initialized in line 60 causes a counterclockwise spinning motion to occur. RO takes on values that result in ROtations that are integral multiples of 90 degrees. (Recall that shapes displayed using ROT values like 8 and 24 appear larger on the screen than those using ROT values like 16 and 32.)

Lines 75-80: Shape 1 in the Arrow shape table is an arrow that points up on the screen; it is

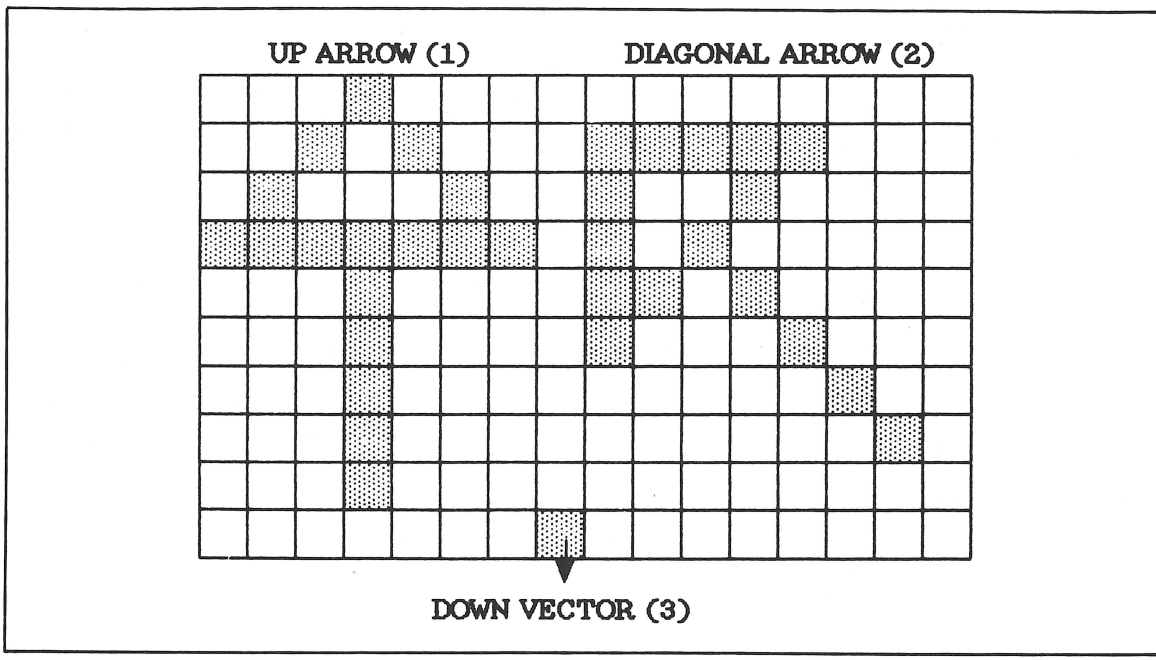


Fig. 8-1. These are the shapes in the Arrows shape table.

Listing 8-7. Arrow Spin I.

```
10  REM     ARROW SPIN I
15  HOME
20  PRINT  CHR$ (4)"BLOAD ARROWS,A$6000"
30  POKE 232,0: POKE 233,96
40  HGR : SCALE= 1
50  FOR REV = 1 TO 10
60  FOR RO = 64 TO 16 STEP  - 16
70  ROT= RO
75  FOR Z = 1 TO 2
80  XDRAW 1 AT 140,70: ON Z GOSUB 1000,1010: NEXT Z
85  FOR Z = 1 TO 2
90  XDRAW 2 AT 140,70: ON Z GOSUB 1000,1010: NEXT Z
100  NEXT RO,REV
110  HGR : TEXT : HOME : END
1000  FOR T = 1 TO 150: NEXT T
1010  RETURN
```

displayed and erased when the statements in lines 75 and 80 are executed. XDRAW causes the shape to be drawn the first time the loop is performed. A time delay subroutine is called after the shape is displayed so that it will not be erased too quickly. The second iteration of the Z loop causes the shape to be erased from the screen. The time loop is not executed after erasing the shape. An ON-GOSUB branch is not performed if the value of the ON variable exceeds the number of branch choices available.

Lines 85-90: Shape 2 in the Arrow shape table is an arrow that represents a 45-degree counterclockwise rotation of shape 1. Lines 85 and 90 are similar to lines 75 and 80. They cause Shape 2 of the Arrow shape table to be displayed and erased.

Lines 100-110: Line 100 causes the rotation loop and the revolution loops to be repeated the required number of times and, after the spinning action has been completed, line 110 clears the screens and ends the program.

Lines 1000-1010: The time delay subroutine used after displaying an arrow is contained in lines 1000-1010. The speed of the spinning motion can be increased or decreased by altering the limit parameter of the loop.

The arrow shapes contained in the Arrow shape table have short shafts. If the arrows needed to be lengthened, the scale value could be increased. However, this change would also increase the size of the arrowhead and would cause the shape to appear distorted when it is displayed on the screen. One way to increase the length of the shaft without affecting the appearance of the shape is to append it to a scaled single-vector shape.

The Arrow Spin II program in Listing 8-8 is similar to the Arrow Spin I program. However, a scaled single-vector shape has been used to extend the length of the shaft.

Three modifications were made to the Arrow Spin I program to obtain the Arrow Spin II program. The changes appear in lines 40, 77, and 87.

Line 40: LS is the variable which is used to scale the single vector shape. The length of the arrow's shaft can be altered by changing the value assigned to LS in line 40.

Line 77: Before the arrow shape is displayed in line 80, the shaft extension is drawn on the screen by line 77. The shaft formed is a scaled single vector. The SCAL value must be set back to 1 after the shaft is displayed. No AT parameter is used with the XDRAW command in line 80, since the arrow shape will be placed at the last point used in drawing the scaled single-vector shape.

Line 87: In order to make the scaled single vector appear to be a part of the arrow displayed in

line 90, it must be rotated 45 degrees counterclockwise. The appropriate values for both SCALE and ROT need to be respecified before the arrow shape is displayed. All this is taken care of in line 87.

Modifications can be made to the Arrow Spin II program so that the color black is used to draw the extended shaft portion of the arrow. As a result, the arrow appears to be orbiting about a point on the screen. Arrow Orbit (Listing 8-9) is a program that demonstrates how to use a scaled single vector to achieve an orbital motion.

Many interesting displays can be created through innovative uses of simple shapes. Wreath (Listing 8-10) is a program that uses shapes 1 and 3 of the Arrow shape table to form a wreath design.

The Wreath program is very similar to the Arrow Orbit program. The shapes are not erased from the screen once they are displayed. Displays like those accomplished with the Wreath program are often achieved accidentally. However, the chances of creating an eye-catching display accidentally improve each time a new technique is attempted. It also helps to be familiar with all the shape-related commands and the color properties of high-resolution graphics.

PLOT GRAPHICS: ENHANCEMENTS OF VECTOR GRAPHICS

The Mixer program presented in Chapter 5 illustrated that shapes and regular high-resolution graphics can be used together to create displays. While most high-resolution commands continue to function as expected when used with shape commands, one command does not.

The HPLOT-TO command can normally be used to extend a line from the most recently plotted point to another point on the screen when the "from" coordinates are omitted. When the most recently plotted point is a result of a DRAW statement, however, the line formed by the HPLOT-TO command does not extend to the point specified as the "to" parameter. The computer uses the coordinates of the last point specified in a DRAW or HPLOT statement to compute the coordinates of the points needed to form the requested line. The resulting line extends to a point offset from the last point plotted the same distance and direction as the

Listing 8-8. Arrow Spin II.

```
10  REM     ARROW SPIN II
15  HOME
20  PRINT  CHR$ (4)"BLOAD ARROWS,A$6000"
30  POKE 232,0: POKE 233,96
40  HGR :LS = 50
50  FOR REV = 1 TO 10
60  FOR RO = 64 TO 16 STEP  - 16
70  ROT= RO
75  FOR Z = 1 TO 2
77  SCALE= LS: XDRAW 3 AT 140,70: SCALE= 1
80  XDRAW 1: ON Z GOSUB 1000,1010: NEXT Z
85  FOR Z = 1 TO 2
87  SCALE= LS: ROT= RO - 8: XDRAW 3 AT 140,70: SCALE=
     1: ROT= RO
90  XDRAW 2: ON Z GOSUB 1000,1010: NEXT Z
100  NEXT RO,REV
105  VTAB 22: PRINT "PRESS ANY KEY.";: GET G$
110  HGR : TEXT : HOME : END
1000  FOR T = 1 TO 150: NEXT T
1010  RETURN
```

Listing 8-9. Arrow Orbit.

```
10  REM      ARROW ORBIT
15  HOME
20  PRINT  CHR$ (4)"BLOAD ARROWS,A$6000"
30  POKE 232,0: POKE 233,96
40  HGR :LS = 50
45  HCOLOR= 0
50  FOR REV = 1 TO 10
60  FOR RO = 64 TO 16 STEP  - 16
70  ROT= RO
75  FOR Z = 1 TO 2
77  SCALE= LS: DRAW 3 AT 140,70: SCALE= 1
80  XDRAW 1: ON Z GOSUB 1000,1010: NEXT Z
85  FOR Z = 1 TO 2
87  SCALE= LS: ROT= RO - 8: DRAW 3 AT 140,70: SCALE= 1
     : ROT= RO
90  XDRAW 2: ON Z GOSUB 1000,1010: NEXT Z
100  NEXT RO,REV
105  VTAB 22: PRINT "PRESS ANY KEY.";: GET G$
110  HGR : TEXT : HOME : END
1000  FOR T = 1 TO 150: NEXT T
1010  RETURN
```

point specified in the HPLOT-TO statement is offset from the point most recently expressed as an HPLOT or DRAW parameter (see Fig. 8-2).

Cylinder (Listing 8-11) is a program similar to the Vector Circle program listed earlier in this chapter. After each vector used in forming the circle is drawn, a line segment is extended from it using the HPLOT-TO command. The resulting design looks like a cylinder or drum. The One Vector Shape table must be on the disk in the drive when the Cylinder program is executed.

Another type of motion sequence that is often useful in action graphics programs is a rocking motion. Rocking motion is similar to spinning motion

Listing 8-10. Wreath.

```
10  REM       WREATH
15  HOME
20  PRINT  CHR$ (4)"BLOAD ARROWS,A$6000"
30  POKE 232,0: POKE 233,96
40  HGR : HCOLOR= 0
60  FOR RO = 0 TO 63
70  ROT= RO + 16
75  SCALE= 35: DRAW 3 AT 140,70
76  ROT= RO
80  SCALE= 1: XDRAW 1
100  NEXT RO
105  VTAB 22: PRINT "PRESS ANY KEY.";: GET G$
110  HGR : TEXT : HOME
```

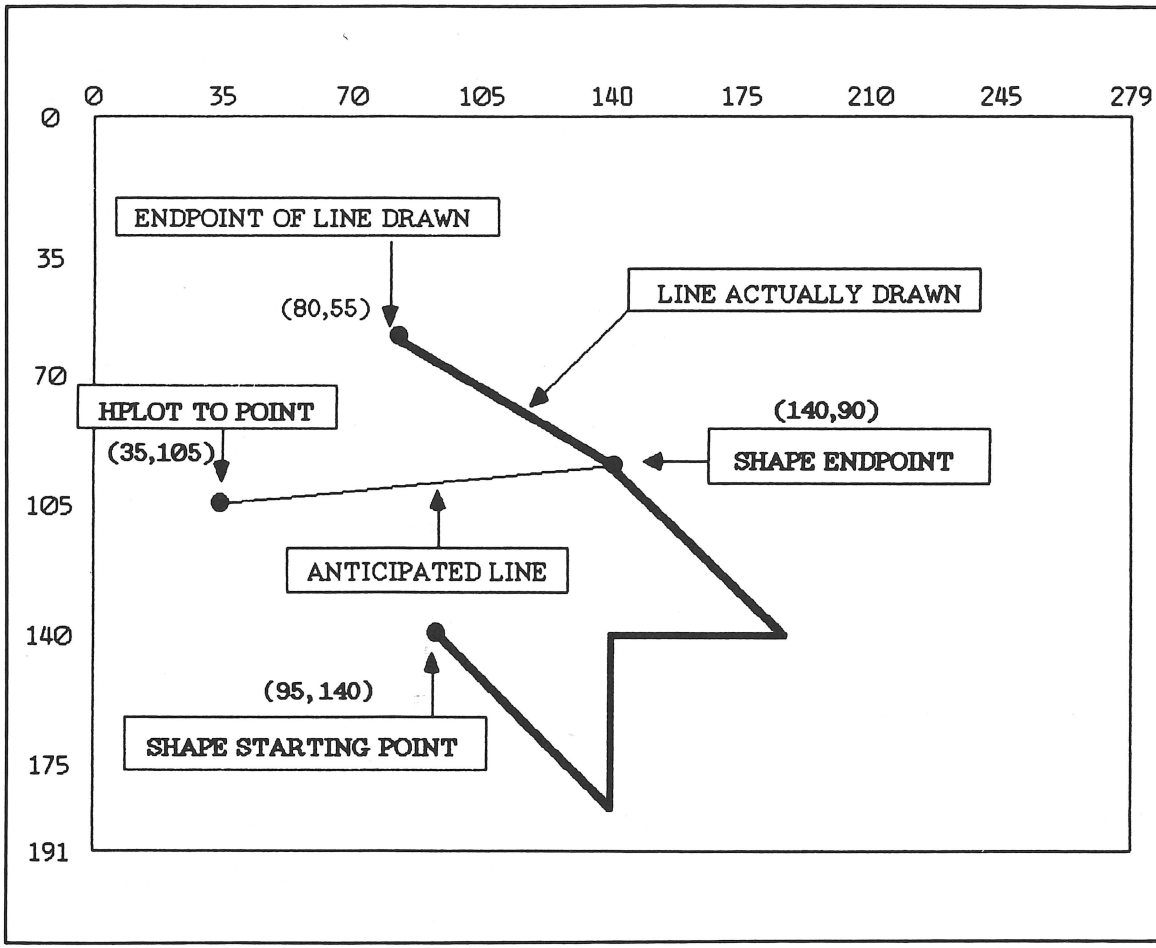


Fig. 8-2. The HPLOT-TO command (with the "from" parameter omitted) works differently when used after the DRAW command than when used after the HPLOT command.

Listing 8-11. Cylinder.

```
10  REM      CYLINDER
15  HOME
20  PRINT  CHR$ (4)"BLOAD ONE VECTOR,A$300"
30  POKE 232,0: POKE 233,3
40  HGR : SCALE= 40: HCOLOR= 3
60  FOR RO = 0 TO 64
70  ROT= RO
80  DRAW 1 AT 140,70
90  HPLOT  TO 170,90
100  NEXT RO
105  VTAB 22: PRINT "PRESS ANY KEY.";: GET G$
110  HGR : TEXT : HOME
```

in that it consists of a spinning motion of one revolution or less, followed by another spinning motion in the opposite direction. A rocking motion can be used to simulate the path of a windshield wiper or pendulum.

Cylinder Sweep (Listing 8-12) is a program that demonstrates a technique for achieving a basic rocking or sweeping motion. The program displays a cylinder design much like that made by the Cylinder program. The end or cap of the cylinder is erased with a counterclockwise sweeping motion. The process is repeated 20 times. After each revolution is completed, the direction of rotation is reversed by the statements in line 55. A half circle rocking motion can be achieved by changing line 55 to:

```
55 F=32-F: L=32-F
```

UTILIZING POLAR EQUATIONS, SHAPES, AND SOUNDS

Nearly every technique presented in this book can be utilized within a single program. Programs are presented in this section that demonstrate the types of displays and effects that can be achieved by drawing from several techniques.

Fan-O-Rama (Listing 8-13) utilizes both the One Vector shape table and the Note Player machine language sound routine. Enter and run the program, observing the visual and sound effects, before reading the discussion of the program listing.

Lines 30-50: These lines load the Note Player machine language routine and One Vector shape table into memory. Care must be taken when selecting the memory locations of the binary files so that they do not overlap and that they do not extend into reserved areas of memory. In line 50 the address of the One Vector shape table is placed in memory locations $00E8 and $00E9.

Line 60: The Note Player program expects to find the length or duration of each note it is to play in memory location 769. All the notes used in the Fan-O-Rama program are the same length (5); the desired value needs to be passed to the program only once, by the POKE statement in line 60. This should be accomplished before the display-creating loops are executed, because each irrelevant statement placed inside the display loop reduces the rate of display. The pitch variable (P) is initialized to 20, but it will be changed within the display loops.

Line 70: High-resolution graphics page 1 is used to display the Fan-O-Rama action design. The line that appears to be rotating across the screen is actually formed by drawing two scaled vectors end-to-end. A scale value of 40 is used to draw each segment. Since the scale value is constant, it can be initialized before the display loop is executed.

Line 80: The variable C represents the col-

Listing 8-12. Cylinder Sweep.

```
10  REM     CYLINDER SWEEP
15  HOME
20  PRINT  CHR$ (4)"BLOAD ONE VECTOR,A$300"
30  POKE 232,0: POKE 233,3
40  HGR : SCALE= 40: HCOLOR= 3
50  FOR REV = 1 TO 20
55 F = 64 - F:L = 64 - F
60  FOR RO = F TO L STEP  SGN (L - F)
70  ROT= RO
80  XDRAW 1 AT 140,70
90  HPLOT  TO 120,50
100  NEXT RO,RE
105  VTAB 22: PRINT "PRESS ANY KEY.";: GET G$
110  HGR : TEXT : HOME
```

Listing 8-13. Fan-O-Rama program.

```
10  REM      FAN-O-RAMA
20  HOME
30  PRINT  CHR$ (4)"BLOAD NOTE PLAYER,A$303"
40  PRINT  CHR$ (4)"BLOAD ONE VECTOR,A$6000"
50  POKE 232,0: POKE 233,96
60  POKE 769,5:P = 20
70  HGR : SCALE= 40:D =  - 1
80 C = 17:F = 64
90  FOR TI = 1 TO 2
100 F = 64 - F:L = 64 - F:SI =  SGN (L - F)
110  FOR Z = 1 TO 2:D =  - D
120  FOR RO = F TO L STEP SI
130 P = P + 1 * D: POKE 768,P: CALL 771
140  IF GP < P THEN GP = P
150 C = C + 3 * D
160  ROT= RO
170  XDRAW 1 AT 20 + C,70: ROT= 32 + RO: XDRAW 1 AT 20
      + C,70
180  NEXT RO:C = C + 3 * D: NEXT Z,TI
190  HGR : TEXT : HOME
```

umn coordinate that is used for placing the scaled vectors on the screen. It is initialized in line 80 to the value 17 to prevent the scaled vector shape from extending beyond the horizontal limits of the screen. F is used to store the ROT value for displaying the vectors in the First horizontal position of each pass across the screen. F is initialized to allow for counterclockwise rotation, but it is changed in line 100 before the first vector shapes are drawn. Therefore the action sequence will begin with clockwise rotation. D is the variable that stores the value, either +1 or −1, for the Direction of motion across the screen. It is initialized for negative (left-to-right) motion, but is switched to cause positive (right-to-left) motion before any shapes are displayed.

Line 90: The TI loop controls the number of TImes the rotating line moves across the screen and back. The first time the loop is executed, the design is placed on the screen. The next execution of the loop removes the design.

Line 100: While the design is being placed on the screen, the scaled vectors are rotated clockwise. The direction of rotation is reversed as the design is erased from the screen. SI is the variable that keeps track of the direction of rotation. The values of the variables F and L are reversed each time this line is executed.

Line 110: In order to create the complete design formed by this program, the motion sequence must be executed twice. The first time this loop is executed, left-to-right movement is accomplished. The second iteration of the loop accounts for the right-to-left movement. The direction of the motion is reversed by changing the value of the Direction variable (D).

Line 120: RO stores the value of rotation for displaying the scaled vector shape used in creating the design. The loop causes the shape to be rotated through one full revolution, either clockwise or counterclockwise.

Line 130: The variable P stores the Pitch of each note played in the program. The pitch variable increases as the spinning vectors move to the right and decrease as they move to the left. The higher the pitch value, the lower the pitch of the note. Line 130 places the pitch value in memory location 768, and then calls the machine language program that plays the note.

Lines 150-160: The column coordinate for

displaying the scaled vector is changed by 3 each time the statement in line 150 is executed. It decreases or increases depending upon the value of D. The rotation value stored in the variable RO is set in line 160.

Line 170: This line displays or erases two scaled single-vector shapes on the screen. Each vector is drawn from the same point, but at 180-degree rotations from one other. The result of this line is a display that appears to be a single line segment with a midpoint at (20+C,70). The XDRAW statement is used to display the vector line segments so that the first time the TI loop is executed the lines will be placed on the screen, and the second time it is used the lines will be erased.

Lines 180-190: After a revolution of the scaled vector line segment is completed, the column coordinate is adjusted in line 180 so the next iteration of the Z loop will cause the rotation cycle to begin displaying line segments at the same column coordinate as the previous execution of the loop. Line 190 clears the screen and displays the text screen.

All movements of shapes that have been demonstrated to this point have consisted of moving the shape from one side of the screen to the other along a horizontal path. Straight paths were used to simplify the motion algorithms. Once the techniques for creating the illusion of motion are known, they can be applied to move shapes along any path. Many of the mathematical formulas used in Chapter 3 for forming high-resolution displays can be used to define a path for a motion sequence.

Mathematical formulas can also contribute to the creation of interesting vector graphics displays. Shapely Ribbon (Listing 8-14) is a program that displays a scaled single-vector shape at points along the path described by the polar equation

R=2*COS(5*T)

where T is allowed to take on values from 0 to 3.14. The program is very much like the Five Petal Rose program presented in Chapter 3.

The Directed Rose program (Listing 8-15) uses shapes, high-resolution graphics, sound, page flipping, and polar equations to form its display. A shape is directed around the three-petal rose path

Listing 8-14. Shapely Ribbon.

```
10  REM     SHAPELY RIBBON
20  PRINT   CHR$ (4)"BLOAD ONE VECTOR,A$6000"
30  POKE 232,0: POKE 233,96
40 S = 1: SCALE= 10: ROT= 40
50  HOME
60  HGR : HCOLOR= 3
70 HS = 30:VS = 30
80 H = 140
90 K = 80
100 TP = 3.14
110 C = 300
120  FOR T = 0 TO TP STEP TP / C
130 R = 2 *  COS (5 * T)
140 X =  COS (T) * R * HS
150 Y =  SIN (T) * R * VS
160  IF  ABS (X) > H OR  ABS (Y) > K THEN 180
170  DRAW S AT H + X,K - Y
180  NEXT T
190  VTAB 22: PRINT "PRESS ANY KEY.";: GET G$
200  HGR : TEXT : HOME
```

Listing 8-15. Directed Rose.

```
10  REM        DIRECTED ROSE
20  PRINT  CHR$ (4)"BLOAD NOTE PLAYER"
30  PRINT  CHR$ (4)"BLOAD GAME SHAPES,A$6000"
40  POKE 769,5
50 S = 3
60  SCALE= 1: ROT= 0
70  HGR : HGR2 : POKE  - 16302,0
80 PG = 1
90  POKE 232,0: POKE 233,96
100  POKE 230,PG * 32
110  FOR T = 0 TO 3.14 STEP .01
120 LC = C:LR = R
130 RA = 2 *  COS (3 * T)
140 C = 45 * RA *  COS (T) + 140
150 R =  - 45 * RA *  SIN (T) + 90
160  HCOLOR= 3
170  DRAW S AT C,R
180  POKE PG - 16301,0
190  GOSUB 280
200 PG = 3 - PG
210  POKE 230,PG * 32
220  HCOLOR= 0
230  DRAW S AT LC,LR
240  IF T THEN  HCOLOR= 3: HPLOT LC,LR
250  NEXT T
260  FOR Z = 1 TO 1000: NEXT Z: HGR2 : TEXT : END
270  REM   >> PLAY A NOTE <<
280  POKE 768,R
290  CALL 771: RETURN
```

defined by the polar equation R=2*COS(3*T). A note is played as the shape is placed at each position on the path. The row coordinate used in drawing the shape is used for the pitch of the note. After a shape is drawn, the high-resolution pages are flipped, the shape is erased, and a single high-resolution point is plotted at that position. The shape appears to move around the screen, pulling a string of dots behind it.

A list of the variables used in the Directed Rose program and their roles is provided below, but a detailed discussion is not provided since no new techniques are introduced.

- S — The Shape which is moved around the screen.
- PG — The high-resolution PaGe is used.
- C — The Column coordinate used for displaying shapes.
- R — The Row coordinate for drawing shapes. R is also used to regulate the pitch of the notes played throughout the program.
- LC — Last Coordinate used for displaying a shape. Since page flipping is used, it needs to be stored so that it can be used in erasing the figure.
- LR — Last Row coordinate used for placing figures on the screen. The program uses LC and LR as coordinates whenever a figure is to be removed from the screen. They are also used to plot the points which form the three-petal rose.
- RA — Distance, or RAdius, from the center of

the three-petal rose to a point on the design.

T Polar angle coordinate used to determine the path the shape will follow on the screen.

Z A time delay loop variable used to leave the completed design on the screen momentarily before it is erased.

The final program in this section, Fraction Action, is an educational drill and practice program that can be used to study additon of fractions. It uses shaded circles to represent fractions that are to be added by the user. Many of the concepts discussed in this book have been incorporated in the program: sound, high-resolution graphics, and shape table text representation. User instructions are presented in the program, presented as Listing 8-16.

Due to the length of the Fraction Action program, the discussion that follows concentrates on

Listing 8-16. Fraction Action educational game program (continues through page 161).

```
10  REM     FRACTION ACTION
20  REM     ===============
25  REM
30  HOME : VTAB 10: HTAB 13: PRINT "FRACTION ACTION"
40  FOR X = 1 TO 10: READ R$(X): NEXT X
50  FOR X = 1 TO 10: READ W$(X): NEXT X
60 D$ =  CHR$ (4): POKE 232,0: POKE 233,96
70  PRINT D$"BLOAD CHARACTER SET,A$6000"
80  PRINT D$"BLOAD LASER UP,A$32D"
90  PRINT D$"BLOAD LASER DOWN"
100  PRINT D$"BLOAD NOTE PLAYER,A$356"
110 HL = 275:VL = 187:I = 8:SP = 3: ROT= 0: SCALE= 1
120  DIM SR$(35):CURS = 59::BLANK = 60: HGR2
130  GOTO 530
135  REM
140  REM  >>> HI-RES PRINT <<<
145  REM
150 ROW = SP + (DR - 1) * I:COLUMN = SP + (DC - 1) * I
160  HCOLOR= SP
170  FOR X = 1 TO  LEN (CH$)
180  IF  MID$ (CH$,X,1) =  CHR$ (32) THEN 200
190  DRAW  ASC ( MID$ (CH$,X,1)) - 32 AT COLUMN,ROW
200 COLUMN = COLUMN + I:DC = DC + 1
210  NEXT X: RETURN
215  REM
220  REM >>> HI-RES INPUT <<<
225  REM
230 COLUMN = SP + (DC - 1) * I:ROW = SP + (DR - 1) * I
    :RC = 1
240 REPLY$ = " ": HCOLOR= SP
250  DRAW CURS AT COLUMN,ROW: GET CH$: XDRAW CURS AT C
    OLUMN,ROW
260  IF CH$ = "" THEN 250
270 CH =  ASC (CH$)
280  IF CH > 32 AND CH < 91 THEN  GOSUB 440: DRAW CH -
    32 AT COLUMN,ROW: GOTO 340
```

```
290  IF CH = 13 THEN 450
300  IF CH = 21 THEN 340
310  IF CH = 32 THEN  GOSUB 440: GOTO 340
320  IF CH = 8 THEN  GOSUB 380
330  GOTO 250
340 RC = RC + 1:COLUMN = COLUMN + I
350  IF COLUMN > HL THEN COLUMN = SP:ROW = ROW + I
360  IF ROW > VL THEN RC = RC - 1: GOTO 450
370  GOTO 250
380  IF ROW > SP + (DR - 1) * I THEN 410
390  IF COLUMN > SP + (DC - 1) * I THEN RC = RC - 1:CO
     LUMN = COLUMN - I
400  GOTO 430
410  IF COLUMN > SP THEN COLUMN = COLUMN - I:RC = RC -
     1: GOTO 430
420 ROW = ROW - I:COLUMN = HL:RC = RC - 1
430  RETURN
440  HCOLOR= 0: DRAW BLANK AT COLUMN,ROW: HCOLOR= SP:S
     R$(RC) = CH$: RETURN
450  FOR X = COLUMN TO HL STEP I: HCOLOR= 0: DRAW BLAN
     K AT X,ROW: NEXT X: HCOLOR= SP
460  FOR X = 1 TO RC - 1
470 REPLY$ = REPLY$ + SR$(X):SR$(X) = ""
480  NEXT X:
490 REPLY$ =  MID$ (REPLY$,2)
500 DC = DC +  LEN (REPLY$)
510  RETURN
515  REM
520  REM >>> POINTS ON CIRCLE <<<
525  REM
530  DIM CZ(300),RZ(300)
540  TEXT : HOME :RS = 40: POKE 769,3
550  GOSUB 1680
560  FOR X = 0 TO 6.29 STEP .02446:P = P + 1
570 CZ(P) = RS * 1.05 *  COS (X):RZ(P) = RS *  SIN (X)
580  IF  PEEK ( - 16384) > 127 THEN  POKE  - 16368,0:Z
     P = ZP + 1: ON ZP GOSUB 1710,1750,1780
590  IF ZP > 2 THEN  POKE 768, RND (1) * 100 + 60: CALL
     854
600  NEXT X
610  IF ZP < 2 THEN  GET G$:ZP = ZP + 1: ON ZP GOSUB 1
     710,1750,1780,1790: GOTO 610
620  HTAB 1: VTAB 23: PRINT "PRESS ANY KEY TO BEGIN.";
     : GET G$: HOME
625  REM
630  REM >>>   MAIN PROGRAM   <<<
635  REM
640  INPUT "HOW MANY PROBLEMS (1-50)? ";NP
650  IF  NOT NP THEN  VTAB  PEEK (37): CALL  - 868: GOTO
     640
660  IF NP > 50 THEN  VTAB  PEEK (37): CALL  - 868: GOTO
```

```
     640
670  FOR Y = 1 TO NP
680 CM = 44:RM = 52
690  GOSUB 1260:S = N1:D = D1
700  HGR2
710 CH$ = F$(1):DC = 5:DR = 1: GOSUB 150
720  GOSUB 1060
730 CH$ = "+":DC = 12: GOSUB 150:CH$ = F$(2):DC = 17: GOSUB
     150
740 D = D2:S = N2
750 CM = CM + 96
760  GOSUB 1060
770 CM = CM + 96
780 DC = 24:CH$ = "=": GOSUB 150
790  GOSUB 1550
800  IF TT > 0 THEN D = 1: GOSUB 1040:RM = RM + 92
810 S = SN:D = CD
820  IF S = 0 THEN D = 1
830  GOSUB 1060
840  GOSUB 1230
850 DC = 1:DR = 23:CH$ = "PRESS ANY KEY TO CONTINUE.":
      GOSUB 150: GET G$: HGR2 : NEXT Y
860 CH$ = "YOU GOT " +  STR$ (CA) + " OUT OF " +  STR$
     (NP) + " RIGHT!"
870 DC = 3:DR = 3: GOSUB 150
880 SN = CA:CD = NP: GOSUB 1630:CH$ = "YOUR SCORE CAN
     BE REPRESENTED":DC = 1:DR = 19: GOSUB 150
890 CH$ = "BY THE FRACTION " +  STR$ (SN) + "/" +  STR$
     (CD) + ".":DC = 5:DR = 21
900  IF SN = CD THEN CH$ = "BY THE NUMBER 1."
910  IF SN = 0 THEN CH$ = "BY A BIG FAT ZERO."
920  GOSUB 150
930  IF SN = 0 THEN S = SN:D = CD:RM = 80:CM = 130: GOSUB
     1060: GOTO 960
940 S = SN:D = CD:RM = 80:CM = 130: GOSUB 1060
950  IF SN / CD > .89 THEN  GOSUB 1010
960 DC = 1:DR = 23:CH$ = "PRESS ANY KEY TO CONTINUE.":
      GOSUB 150
970  GET G$
980  HGR2 : TEXT : HOME : INPUT "DO YOU WANT TO PRACTI
     CE SOME MORE? ";G$: IF  LEFT$ (G$,1) = "N" THEN  END
990 CA = 0: GOTO 640
995  REM
1000  REM >>> HIGH SCORE SOUND <<<
1005  REM
1010  POKE 773,CA: POKE 775,1: POKE 780,255: CALL 772
1020  POKE 773,3 * NP: POKE 775,70: POKE 780,150: CALL
     772
1030  RETURN
1040 S = D: GOSUB 1060: RETURN
1050  REM  >>>  DRAW CIRCLE  <<<
```

```
1060  HCOLOR= SP: FOR X = 1 TO P: HPLOT CZ(X) + CM,RZ(
      X) + RM: NEXT X
1065  REM
1070  REM >>>  DIVIDE CIRCLE  <<<
1075  REM
1080  IF D < 2 THEN 1130
1090  HCOLOR= SP: FOR X = 1 TO P STEP P / D
1100  HPLOT CM,RM TO CZ(X) + CM,RZ(X) + RM
1110  NEXT X
1115  REM
1120  REM >>>  SHADE SECTIONS  <<<
1125  REM
1130  IF S * (P / D) < 1 THEN  GOTO 1220
1140  HCOLOR= 1: FOR X = 1 TO S * (P / D)
1150  HPLOT CM,RM TO CZ(X) + CM,RZ(X) + RM
1160  NEXT X
1165  REM
1170  REM  >>>  DIVIDE CIRCLE  <<<
1175  REM
1180  IF D < 2 THEN 1220
1190  HCOLOR= SP: FOR X = 1 TO P STEP P / D
1200. HPLOT CM,RM TO CZ(X) + CM,RZ(X) + RM
1210  NEXT X
1220  RETURN
1230 DR = 1:DC = 29:CH$ = F$(3): GOSUB 150
1240  RETURN
1245  REM
1250  REM  >>> INPUT PROBLEM <<<
1255  REM
1260  TEXT : HOME : PRINT
1270  INPUT "FIRST FRACTION: ";F$(1):N1 =  VAL (F$(1))
1280 D1 = 0:KS = 0
1290  FOR Z = 1 TO  LEN (F$(1)): IF  MID$ (F$(1),Z,1) =
      "/" THEN D1 =  VAL ( MID$ (F$(1),Z + 1)):KS = KS +
      1
1300  NEXT Z: IF KS <  > 1 THEN  VTAB  PEEK (37): CALL
       - 868: GOTO 1270
1310  IF D1 = 0 THEN  PRINT "THE DENOMINATOR CANNOT BE
      0 - TRY AGAIN.": GOTO 1270
1320  IF N1 > D1 THEN  PRINT "USE PROPER FRACTIONS ONL
      Y, PLEASE.": GOTO 1270
1330  INPUT "SECOND FRACTION:  ";F$(2):N2 =  VAL (F$(2
      ))
1340 D2 = 0:KS = 0
1350  FOR Z = 1 TO  LEN (F$(2)): IF  MID$ (F$(2),Z,1) =
      "/" THEN D2 =  VAL ( MID$ (F$(2),Z + 1)):KS = KS +
      1
1360  NEXT Z: IF KS <  > 1 THEN  VTAB  PEEK (37): CALL
       - 868: GOTO 1330
1370  IF D2 = 0 THEN  PRINT "THE DENOMINATOR CANNOT BE
      0 - TRY AGAIN.": GOTO 1330
```

```
1380  IF N2 > D2 THEN  PRINT "USE PROPER FRACTIONS ONL
     Y, PLEASE.": PRINT : GOTO 1330
1390 K1 = 1:K2 = 1:C1 = D1:C2 = D2
1400  IF C1 < C2 THEN K1 = K1 + 1:C1 = D1 * K1: GOTO 1
     400
1410  IF C2 < C1 THEN K2 = K2 + 1:C2 = D2 * K2: GOTO 1
     410
1420  IF C1 <  > C2 THEN 1400
1430 CD = C1:SN = N1 * K1 + N2 * K2
1440  PRINT
1450  IF CD > 99 THEN  PRINT  CHR$ (7);"THAT'S TOO TOU
     GH FOR ME!": FOR Z1 = 1 TO 1500: NEXT Z1: GOTO 12
     60
1460 TT = 0
1470  IF SN > CD THEN TT = TT + 1:SN = SN - CD
1480  GOSUB 1630
1490 F$(3) =  STR$ (SN) + "/" +  STR$ (CD)
1500  IF SN = CD THEN F$(3) =  STR$ (TT + 1): IF  VAL
     (F$(3)) > 1 THEN 1530
1510  IF TT > 0 THEN F$(3) =  STR$ (TT) + " " + F$(3)
1520  IF SN = 0 THEN F$(3) = "0"
1530  RETURN
1535  REM
1540  REM  >>> GET ANSWER <<<
1545  REM
1550 CH$ = "WHAT IS THE ANSWER?  "
1560 DC = 1:DR = 13: GOSUB 150: GOSUB 230:DR = 16:DC =
     1
1570 ZZ = 2 * ( INT ( RND (1) * 5)) + 1
1580  POKE 773,1: POKE 775,1: POKE 780,255: POKE 814,1
     : POKE 816,1: POKE 821,255
1590  IF REPLY$ <  > F$(3) THEN CH$ = W$(ZZ): CALL 813
     : GOSUB 150:CH$ = W$(ZZ + 1)
1600  IF REPLY$ = F$(3) THEN CH$ = R$(ZZ):CA = CA + 1:
      CALL 772: GOSUB 150:CH$ = R$(ZZ + 1)
1610 DC = 1:DR = 18: GOSUB 150: RETURN
1615  REM
1620  REM  >>> REDUCE FRACTION <<<
1625  REM
1630  FOR RD = 2 TO SN
1640  IF SN / RD =  INT (SN / RD) AND CD / RD =  INT (
     CD / RD) THEN SN = SN / RD:CD = CD / RD: GOTO 163
     0
1650  NEXT RD
1660  RETURN
1665  REM
1670  REM  >>> INSTRUCTIONS <<<
1675  REM
1680  HOME : PRINT : PRINT "I AM GOING TO HELP YOU STU
     DY FRACTIONS.": PRINT : PRINT "YOU NEED TO TELL M
     E TWO FRACTIONS TO ": PRINT : PRINT "ADD.  I WILL
```

```
      DRAW CIRCLES AND COLOR": PRINT : PRINT "THE AREA
     S THAT REPRESENT THOSE": PRINT : PRINT "FRACTIONS
     ."
1690  PRINT : PRINT : PRINT "    I WILL THEN ASK YOU F
     OR THE ANSWER.": PRINT : PRINT "I WILL TELL YOU I
     F YOU ARE RIGHT OR ": PRINT : PRINT "WRONG, THEN
     DRAW A CIRCLE OR CIRCLES": PRINT : PRINT "REPRESE
     NTING THE CORRECT ANSWER."
1700  VTAB 23: PRINT "PRESS ANY KEY TO CONTINUE.";: RETURN

1710  HOME : PRINT : PRINT "MAKE SURE YOU ONLY GIVE ME
      PROPER": PRINT : PRINT "FRACTIONS (SMALLER TOP T
     HAN BOTTOM).": PRINT : PRINT "TELL ME YOUR FRACTI
     ONS LIKE THIS:": PRINT : PRINT : PRINT : PRINT "
                    1/3": PRINT : PRINT : PRINT
1720  PRINT "WITH A SLASH BETWEEN THE TOP AND BOTTOM":
      PRINT : PRINT "NUMBERS OF THE FRACTION."
1730  PRINT : PRINT "YOU WILL BE ALLOWED TO SAY HOW MA
     NY": PRINT : PRINT "PROBLEMS YOU WANT TO DO."
1740  VTAB 23: PRINT "PRESS ANY KEY TO CONTINUE.";: RETURN

1750  HOME : PRINT "YOU MUST REDUCE YOUR ANSWERS TO LO
     WEST": PRINT : PRINT "TERMS OR I WILL TELL YOU TH
     EY ARE WRONG.": PRINT : PRINT "IF THE ANSWER IS A
     N IMPROPER FRACTION,": PRINT : PRINT "YOU MUST GI
     VE IT AS A MIXED NUMBER.": PRINT
1760  PRINT "TO TELL ME A MIXED NUMBER,": PRINT : PRINT
     "SPACE BETWEEN THE WHOLE NUMBER AND THE": PRINT :
      PRINT "FRACTION LIKE THIS:": PRINT : PRINT : PRINT
     "          1 7/8": PRINT : PRINT "GOOD LUCK!"
1770  VTAB 23: PRINT "PRESS ANY KEY TO BEGIN.";: RETURN

1780  HOME : PRINT : PRINT "I'M NOT QUITE READY -- PLE
     ASE WAIT."
1790  RETURN
1800  POKE 768,150 - ( PEEK (768)): POKE 769,20: CALL
     854: RETURN
1810  DATA  "I GOT THE SAME ANSWER -","      SEE?
      ","****** GREAT GUNS ******","I THINK YOU'VE GOT
      IT!","THAT'S KIND OF THE WAY","I HAD IT FIGURED.
     ","AS THEY SAY ON FAMILY","FEUD - GOOD ANSWER!","
     THAT'S REALLY","WONDERFUL!!!"
1820  DATA  "YOU'LL NEVER CONVINCE ME.","LOOK WHAT I G
     OT!","NOT A CHANCE.  HOW DID","YOU EVER GET THAT?
     ","YOU MUST BE KIDDING!","IF NOT YOU SHOULD BE.",
     "SORRY, BUT YOU'RE NOT","RIGHT THIS TIME."
1830  DATA  "NOT UNLESS IT'S JULY","AND IT'S SNOWING!"
```

the functions of the major sections of the program.

Lines 40-50: After the student enters an answer for a fraction addition problem, the computer responds with either a positive or negative reply. Replies to correct answers are stored in the string array named R$. Replies to incorrect re-

sponses are stored in the array named W$. Each array stores five unique replies that are randomly selected. The two lines of each reply are stored in consecutive elements of the arrays.

Lines 60-100: The binary files needed in the program are loaded into appropriate memory segments. The address of the Character Set shape table is stored at $00E8 and $00E9 in line 60. All the sound-generating machine language routines used in the program are placed in memory.

Lines 110-120: The variables used in the routines that display text characters on the high-resolution screen are assigned initial values in these lines. The routines used for displaying text on the graphics screen are those developed for the Mixer program in Chapter 5.

Line 130: The subroutines that display text characters on the high-resolution screen are situated early in the program to minimize the time required to call them. Line 130 causes the program to branch around those routines.

Lines 150-210: This subroutine is used to display text characters on the high-resolution screen. The same subroutine was used in the Mixer program.

Lines 230-510: The subroutine that is used to input answers and echo those responses to the high-resolution graphics screen is contained in this section of the program. A BAD SUBSCRIPT error occurs if the user's response exceeds 35 characters. Longer responses can be accommodated by increasing the number of elements dimensioned for the SR$( ) array in line 120.

Lines 530-620: Circles, all of which have the same radius, are used extensively by this program. If the coordinates of the points used in constructing the circles had to be calculated each time a circle was displayed, the program would be slow and boring. In order to decrease the time required to draw the circles, the calculations are performed before the drill begins and the results stored in integer arrays. The row coordinates are stored in the R%( ) array, and the column coordinates are stored in the C%( ) array.

Three pages of instructions, each stored in a separate subroutine, are presented to the student by the program. In order to make the best use of time, the instructions can be read while the points on the circle are being calculated. The first page of instructions is displayed when line 550 is executed. After the subroutine places the instructions on the screen, the program begins calculating the points on the circle.

After the first page has been read, the student tells the program to display the second page. The calculations are interrupted just long enough to display the instructions, and then are immediately resumed. If all three pages of instructions are read before the completion of the coordinate calculation loop, the program begins playing random notes (via the note player machine language routine) until the calculations are completed.

The radius used for the circle calculations is assigned to the variable RS. The number of points used to form the circle is counted by the variable P. ZP is the variable used to keep track of the number of instruction pages that have been displayed.

Lines 640-650: The drill begins by asking the student the number of problems he or she wants to do. The number of problems desired is assigned to the variable NP. Any number of problems from 1 through 50 can be requested.

Lines 670-850: This loop constitutes the main part of the program. The loop calls subroutines that obtain the addition problem, display the statement and illustration of the problem on the high-resolution graphics screen, obtain the student's answer to the problem, and then check it for accuracy. Each line in this loop is discussed below.

Line 680: The values assigned to CM and RW in line 680 are the coordinates for the center of the circle used to represent the first fraction addend.

Line 690: Line 690 calls the subroutine that asks the student to input an addition problem. After returning from the subroutine, the numerator of the first fraction (N1) given is assigned to the variable S, and the denominator (D1) is assigned to the variable D. S and D are used by the subroutines that divide and shade the circles that represent the addition problem and its solution.

Lines 700-710: Page 2 of high-resolution graphics is displayed in line 700. The first fraction of

the addition problem (F$(1)) is assigned to CH$. The characters stored in CH$ are placed on the screen starting at display column (DC) 5 and display row (DR) 1 when the high-resolution output subroutine at line 150 is called.

Line 720: Subroutine 1060 causes a fraction to be represented graphically. It draws a circle, divides it into D sections, and shades S sections of the circle. When line 720 calls the subroutine, the first fraction of the problem will be drawn.

Line 730: When this line is executed, the rest of the addition problem is placed on the screen. The subroutine in line 150 is called twice. The first time it is called, the plus sign is placed on the screen. The second time, the second fraction of the addition problem is written.

Lines 740-760: These lines cause the circle which represents the second fraction of the addition problem to be placed on the screen.

Lines 770-780: In order to complete the statement of the addition problem, an equal sign is placed to the right of the second fraction.

Line 790: This line calls the subroutine that obtains the answer to the addition problem. Subroutine 1550 also checks the answer and outputs an appropriate remark to indicate if the student was right or wrong.

Lines 800-840: These lines display the circle(s) that represent the solution to the addition problem. The instructions in line 800 are only performed if the answer is greater than one, since they cause a solid circle to be displayed. If two circles are needed to represent the answer, the second circle is placed directly below the first circle when lines 810 through 830 are executed. Line 840 writes the solution at the top of the screen, to the right of the equal sign.

Line 850: The last line of the main loop of the program places a prompt at the bottom of the screen, instructing the student to press any key to continue with the exercise. The graphics screen is cleared before the loop repeats, so the current problem will not flash on the screen before the next problem is displayed.

Lines 860-990: This section of the program gives the student a grade. It outputs the number of problems answered correctly, the number of problems attempted, the fraction which describes the score, and a shaded circle or circles that represent that fraction. If the student obtained a score of 90 percent or better, then a sound reward is given. The student is then given the opportunity to practice more addition problems.

Lines 1010-1030: This subroutine causes a series of sounds to be generated, and is used whenever a student performs well on an exercise. The Laser Up machine language program is used to create the sound. The sound generated depends on the number of problems answered correctly (CA) and the number of problems attempted (NP).

Lines 1060-1220: This subroutine draws circles that represent fractions. First, a circle is placed on the screen. Next, it is divided into D (denominator) sections. A portion of the circle representing the numerator of the fraction is then shaded with the color green. Since the shading process causes some of the section lines to disappear, the circle is divided into sections again after it has been shaded.

Lines 1260-1530: The addition problems are entered by the student. The instructions in the program indicate that only proper fractions should be used. However, fractions representing the number 1 may also be entered. Checks are performed to ensure the fraction entered does not have a denominator of zero. Fractions that have a common denominator (CD) greater than 99 are not permitted because they cannot be represented well using high-resolution graphics circles.

After line 1430 is executed, the numerator of the solution is assigned to the variable NS, and the denominator is assigned to the variable CD. The answer is reduced to lowest terms by the subroutine located at line 1630. TT is the variable used to flag solutions greater than one. Once the best form of the answer is determined, it is assigned to the string F$(3) and the subroutine is exited.

Lines 1550-1610: This subroutine obtains the answer to the problem from the student and echos it on the screen. Both the high-resolution text output and the text input subroutines are used to accomplish this task. After the student has entered

the answer, it is compared to the correct answer and an appropriate response is selected randomly to inform the student whether the solution was correct. If the student answers the problem correctly, a sound is generated using the Laser Up machine language program. If the student answers the problem incorrectly, a sound is generated using the Laser Down program.

Lines 1630-1660: The instructions indicate that the student must reduce answers to lowest terms. This subroutine is executed after the computer calculates the answer to ensure that it is, in fact, in lowest terms.

Lines 1680-1790: The instructions concerning the fraction addition drill are contained in subroutines in this section of the program. They are delivered while the program is calculating the points of the circle used to represent the problems. The program allows the student to display the next of three pages of instructions by pressing a key.

Lines 1810-1830: These DATA statements store the computer responses to correct and incorrect answers supplied by the student. Each response is stored in two consecutive elements. The responses are split to prevent them from extending into the portion of the screen needed for displaying the circles for solutions that are greater than one.

The Fraction Action program can be used to help students learn addition of fractions in several ways. One approach is to allow students to think up their own problems. Another way is to provide a list of problems they are to solve. Whichever approach is used, the program can be used for students at various skill levels. It is a suitable drill for adding fractions with the same denominator or different denominators.

Modifications could be made to the program that would allow the computer to select the fractions randomly. Another modification that might be useful is to have the program obtain the problems either from DATA statements or from text files assembled by the instructor. It could even be changed so that any form of the correct answer is accepted—rather than just the form in lowest terms.

9

Machine Language Topics

High-quality programs can be developed in BASIC using the graphics and sound techniques presented in earlier chapters. However, some programs must be executed more quickly than is possible with BASIC. While compiled BASIC programs and programs written in other languages that utilize compilers execute more quickly than standard BASIC, very high speed arcade-quality games are best accomplished using machine language programs.

Teaching machine language is beyond the scope of this book, but several machine language programs are listed in this chapter. These were designed to provide readers the opportunity to compare the execution speed of machine language programs with similar programs written in BASIC. In order to develop truly fast arcade games, the programming must be done in machine language. If you have not wanted to learn machine language programming because you thought it was not worth the trouble, you may change your mind after running the programs in this chapter.

DISPLAYING SHAPES IN MACHINE LANGUAGE

Programs were presented in the previous chapter that demonstrated the difference between page flipping and single-page motion graphics in BASIC. Now let's look at programs that demonstrate the difference between the two techniques in machine language. The Compare Motions machine language program (Listing 9-1) can be used with any shape table loaded in memory at address $6000. The shape table must be loaded before running the program. The statement POKE 232,0 : POKE 233,96 is not needed, since an equivalent statement is included in the machine language program.

Use the statement BRUN COMPARE MOTIONS to run the program directly from the disk. Once the program is in memory, it can be run by typing CALL 770 when the BASIC prompt character (]) is displayed, or by typing 302G when the Monitor prompt character (*) is displayed.

Listing 9-1. Compare Motions assembly language listing (continues through page 169).

```
1                ORG   $302
2     SCALE      EQU   $E7
3     HGR2       EQU   $F3D8
4     HGR        EQU   $F3E2
5     HPOSN      EQU   $F411
6     SHPTR      EQU   $F730
7     DRAW       EQU   $F601
8     SHLOW      EQU   $1A
9     SHHIGH     EQU   $1B
10    LBTBL      EQU   $E8
11    HBTBL      EQU   $E9
12    HCOLOR     EQU   $E4
13    SPEAKER    EQU   $C030
14    PAGE       EQU   $E6
15    PGFLIP     EQU   $C053
16    FULLPG     EQU   $C052
17    USEPG      EQU   $300
18
19    *=========================*
20    *                         *
21    * PAGE FLIPPING MOTION    *
22    *                         *
23    *=========================*
24
25    *  POKE 232,0 : POKE 233,96
26
27               LDA   #$00
28               STA   LBTBL
29               LDA   #$60
30               STA   HBTBL
31
32    *  HGR2 : HGR  *
33
34               JSR   HGR2
35               JSR   HGR
36
37    *  POKE -16302,0  *
38
39               LDA   #$00
40               STA   FULLPG
41
42    *  SCALE=1  *
43
44               LDA   #$01
45               STA   SCALE
46
47    *  PG=2  *
48
49               LDA   #$02
```

```
50              STA  USEPG
51
52    *  POKE 230,64  *
53
54              LDA  #$40
55              STA  PAGE
56
57    *  HCOLOR=3  *
58
59    MOVEIT    LDA  #$7F
60              STA  HCOLOR
61
62    *  R=100  *
63    *  C=5  (FIRST TIME THROUGH)  *
64
65              LDA  #$64
66    HLOW      LDX  #$05
67              LDY  #$00
68              JSR  HPOSN
69
70    *  X=PEEK(49200)  *
71
72              LDA  SPEAKER
73
74    *  SHAPE=1  *
75
76              LDX  #$01
77              JSR  SHPTR
78
79    *  NO BASIC EQUIVALENT  *
80    *  FIND ADDRESS FOR DRAW OR PLOT  *
81
82              LDX  SHLOW
83              LDY  SHHIGH
84
85    *  ROT=0 *
86
87              LDA  #$00
88
89    *  DRAW 1 AT C,R  *
90
91              JSR  DRAW
92
93    *  POKE -16301+PG,,0  *
94
95              LDX  USEPG
96              LDA  #$00
97              STA  PGFLIP,X
98
99    *  X=PEEK(49200)
100
```

```
101             LDA  SPEAKER
102             SEC
103
104   *  PG=3-PG  *
105
106             LDA  #$03
107             SBC  USEPG
108             STA  USEPG
109
110   *  POKE 230,96-PEEK(230)  *
111   *  REM  USE HIDDEN PAGE   *
112
113             LDA  #$60
114             SBC  PAGE
115             STA  PAGE
116
117   *  HCOLOR=0  *
118
119             LDA  #$00
120             STA  HCOLOR
121
122   *  DRAWS SHAPE 1 AT 100,C-1 WITH ROT=0  *
123
124             LDA  #$64
125             LDX  HLOW+1
126             DEX
127             LDY  #$00
128             JSR  HPOSN
129             LDX  #$01
130             JSR  SHPTR
131             LDX  SHLOW
132             LDY  SHHIGH
133             LDA  #$00
134             JSR  DRAW
135
136   *  C=C-1  *
137
138             LDX  HLOW+1
139             INX
140             STX  HLOW+1
141
142   *  IF C<>255 THEN GOTO 59 (MOVEIT)  *
143
144             CPX  #$FF
145             BNE  MOVEIT
146
147   *=========================*
148   *                         *
149   *     PAGE 1 MOTION       *
150   *                         *
151   *=========================*
```

```
152
153  *  THIS PROGRAM IS EQUIVALENT TO:
154  *  ------------------------------
155  *  HGR
156  *  FOR C=5 TO 255
157  *  HCOLOR=3 : X=PEEK(49200)
158  *  DRAW 1 AT C,100
159  *  HCOLOR=0 : X=PEEK(49200)
160  *  DRAW 1 AT C,100 : NEXT C
161
162            JSR  HGR
163            LDA  #$7F
164            STA  HCOLOR
165  FRAME     LDA  #$64
166  HLOW2     LDX  #$05
167            LDY  #$00
168            JSR  HPOSN
169            LDA  SPEAKER
170            LDX  #$01
171            JSR  SHPTR
172            LDX  SHLOW
173            LDY  SHHIGH
174            LDA  #$00
175            JSR  DRAW
176            LDA  HCOLOR
177            EOR  #$7F
178            STA  HCOLOR
179            BEQ  FRAME
180            INC  HLOW2+1
181            BNE  FRAME
182            RTS
```

The hexadecimal listing of the Compare Motions (Listing 9-2) program should be entered from the Monitor starting at memory location $302. The program can be saved on the disk using the statement BSAVE COMPARE MOTIONS, A$302, L175.

The Motion Comparison program (Listing 9-3) is included so that you can compare more easily the visual effects of single-page and page flipping graphics motion. Keep in mind that the algorithm or procedure used to create the displays in machine language are as close as possible to those used to

Listing 9-2. Hexadecimal listing of Compare Motions.

```
302:A9 00 85 E8 A9 60 85 E9 20 D8 F3 20
E2 F3 A9 00 8D 52 C0 A9 01 85 E7 A9 02
8D 00 03 A9 40 85 E6 A9 7F 85 E4 A9 64
A2 05 A0 00 20 11 F4 AD 30 C0 A2 01 20
30 F7

:A6 1A A4 1B A9 00 20 01 F6 AE 00 03 A9
```

The Raster Motion program (Listing 9-4) has three delay loops that cause the computer to count to 768 before it erases the figure from the screen and redraws it. Such delays were not used in the Compare Motions program, but they are needed in Raster Motion so that it will run slowly enough for the figures to be seen. If the delay loops are omitted, the program runs so quickly that the figure is never on the screen long enough to see it. Raster graphics are very fast!

Enter and run Raster Motion. Notice that page flipping is not used. The programming strategies used in the program are not necessarily those that would be used in most programs using raster graphics; the primary consideration is designing the program was to keep it as short as possible.

Listing 9-4. Raster Motion assembly listing.

```
1                ORG   $301
2     HGR        EQU   $F3E2
3     SPEAKER    EQU   $C030
4
5     *  RASTER MOTION--PAGE 1  *
6
7     SETUP      JSR   HGR
8                LDY   #$00
9     MOVE       LDX   #$05
10    PLACE      LDA   FIGURE-1,X
11    SCREEN     STA   $2128
12               CLC
13               LDA   SCREEN+2
14               ADC   #$04
15               STA   SCREEN+2
16               DEX
17               BNE   PLACE
18               LDA   SPEAKER
19    DELAY1     INX
20               BNE   DELAY1
21    DELAY2     INX
22               BNE   DELAY2
23    DELAY3     INX
24               BNE   DELAY3
25               LDA   SPEAKER
26               LDX   #$05
27    ERASE      LDA   #$00
28    BLACKOUT   STA   $2128
29               CLC
30               LDA   BLACKOUT+2
31               ADC   #$04
32               STA   BLACKOUT+2
33               DEX
34               BNE   ERASE
35               INC   SCREEN+1
36               INC   BLACKOUT+1
37               INY
38               CPY   #$28
```

```
39                BEQ   END
40                LDA   #$21
41                STA   SCREEN+2
42                STA   BLACKOUT+2
43                JMP   MOVE
44    END         RTS
45
46    FIGURE      HEX   60,50,48,50,60
```

The hexadecimal listing of Raster Motion (Listing 9-5) should be entered at $301. It can be saved on the disk with the statement BSAVE RASTER MOTION, A$301, L87.

As mentioned previously, the primary purpose of this chapter is to introduce you to the tremendous power of machine language programming. We hope that some of you will now pursue a topic that may have seemed threatening before this exposure. You are reminded, however, that very high quality graphics programs can be developed using the techniques presented in earlier chapters.

Listing 9-5. Hex listing of Raster Motion.

```
301:20 E2 F3 A0 00 A2 05 BD 52 03 8D 28
21 18 AD 0D 03 69 04 8D 0D 03 CA D0 EE
AD 30 C0 E8 D0 FD E8 D0 FD E8 D0 FD AD
30 C0 A2 05 A9 00 8D 28 21 18 AD 2F 03

:69 04 8D 2F 03 CA D0 EF EE 0C 03 EE 2E
03 C8 C0 28 F0 0B A9 21 8D 0D 03 8D 2F
03 4C 06 03 60 60 50 48 50 60
```

Appendix A
ASCII Codes

Each text and control character used by the computer has been assigned a unique code, called its *ASCII code* (American Standard Code for Information Interchange). BASIC has two useful functions which operate on the ASCII code of a character. This appendix gives brief explanations of these functions, as well as an ASCII conversion chart.

The CHR$( ) function evaluates an ASCII code and returns the character the code represents. This function can be used to output text charactres that are not available from the computer keyboard. For example, the underline character cannot be typed from the keyboard on some computers, but the statement PRINT CHR$(95) causes the underline character to be output to the screen. It can also be used to make program listings more readable. The BASIC statement

```
PRINT ""
```

used in a program is not meaningful without supporting remarks statements. Any number of control characters could reside between the quotation marks, since control characters cannot be displayed on the screen. The statement is much easier to understand when the CHR$( ) function is used to express the control characters. Use of the CHR$( ) function to express control characters also makes programs easier to edit. If the statement PRINT "" with a CTRL-G between the quotation marks needs to be edited, the programmer must remember to press the Control and G keys between the quotes. Simply passing the cursor over the statement using the right arrow key causes the control character to be lost. The editing problem is avoided if the statement PRINT CHR$(7) is used to produce the BELL tone.

ASC( ) is another useful function which utilizes ASCII codes. This function performs the opposite operation of the CHR$( ) function: it evaluates a character and returns its ASCII code. For example, the statement PRINT ASC("A") causes the number 65 to be printed. The argument of the ASCII function must be a string value. If the statement ASC(G$) is used in a program, the ASCII code of the first character in the string G$ will be returned.

Several of the programs presented in this book use the CHR$( ) and ASC( ) functions. An ASCII code chart is provided in Table A-1.

Table A-1. ASCII Character Codes.

DECIMAL	HEX	CHARACTER	DECIMAL	HEX	CHARACTER	DECIMAL	HEX	CHARACTER
0	00	NULL	32	20	SPACE	64	40	@
1	01	CTRL-A	33	21	!	65	41	A
2	02	CTRL-B	34	22	"	66	42	B
3	03	CTRL-C	35	23	#	67	43	C
4	04	CTRL-D	36	24	$	68	44	D
5	05	CTRL-E	37	25	%	69	45	E
6	06	CTRL-F	38	26	&	70	46	F
7	07	BELL	39	27	'	71	47	G
8	08	BACK SPC	40	28	(	72	48	H
9	09	CTRL-I	41	29	)	73	49	I
10	0A	CTRL-J	42	2A	*	74	4A	J
11	0B	CTRL-K	43	2B	+	75	4B	K
12	0C	CTRL-L	44	2C	,	76	4C	L
13	0D	RETURN	45	2D	-	77	4D	M
14	0E	CTRL-N	46	2E	.	78	4E	N
15	0F	CTRL-O	47	2F	/	79	4F	O
16	10	CTRL-P	48	30	0	80	50	P
17	11	CTRL-Q	49	31	1	81	51	Q
18	12	CTRL-R	50	32	2	82	52	R
19	13	CTRL-S	51	33	3	83	53	S
20	14	CTRL-T	52	34	4	84	54	T
21	15	CTRL-U	53	35	5	85	55	U
22	16	CTRL-V	54	36	6	86	56	V
23	17	CTRL-W	55	37	7	87	57	W
24	18	CTRL-X	56	38	8	88	58	X
25	19	CTRL-Y	57	39	9	89	59	Y
26	1A	CTRL-Z	58	3A	:	90	5A	Z
27	1B	ESC	59	3B	;	91	5B	[
28	1C	N/A	60	3C	<	92	5C	\
29	1D	CTRL-]	61	3D	=	93	5D	]
30	1E	CTRL-^	62	3E	>	94	5E	^
31	1F	N/A	63	3F	?	95	5F	_

Appendix B

Computer Number Systems

The number systems most often associated with the study of computer programming are *decimal, binary,* and *hexadecimal.* Each number system has a unique base number which is used to determine the place values of the digits used to express the number. A number system is often referenced by its base. Decimal numbers are base-10, binary numbers are base-2, and hexadecimal numbers are base-16.

The number of unique numerals or symbols needed to express all the values of a number system corresponds to the system's base number. Therefore, the base-10 number system requires 10 symbols, base-2 needs 2 symbols, and base-16 needs 16 different symbols for representing numbers. Digits used to represent decimal numbers are 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Binary numbers are represented with the two digits 0 and 1. Hexadecimal numbers are expressed using these symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F.

People generally use the base-10 number system for expressing numbers and performing calculations. The computer uses base-2 and base-16. While the number base conversions are usually performed by the computer, some techniques discussed in this book require the programmer to be able to convert from one number system to another.

The Number Base Conversion program Listing B-1 can be used for converting numbers from bases 2 through 16 to any other number base from 2 through 16. Conversions of numbers above 999,999,999 (base-10) performed by the program will not always be accurate due to limitations of the computer. The program is suitable, however, for any conversions necessary for the techniques discussed in this book.

Number base conversion charts also are provided in those chapters that require extensive conversions between the number systems commonly used with computers.

Listing B-1. Number Base Conversion program.

```
10  REM  NUMBER BASE CONVERSION
20  REM  ====== ==== ==========
30  DIM D$(15)
40  FOR X = 0 TO 15: READ D$(X): NEXT X
50  HOME
60  INPUT "CONVERT TO WHAT BASE? ";BT
70  INPUT "CONVERT FROM WHAT BASE? ";BF
80  INPUT "ENTER A NUMBER TO CONVERT: ";N$
90  PRINT : PRINT
100  REM ----------------------
110  REM    CHANGE TO BASE 10
120  REM ----------------------
130 L =  LEN (N$)
140  FOR X = L TO 1 STEP  - 1
150  FOR Z = 0 TO 15
160  IF  MID$ (N$,X,1) = D$(Z) THEN 180
170  NEXT Z
180  IF Z >  = BF THEN  PRINT N$" IS NOT A BASE "BF" N
     UMBER.": END
190 DEC = DEC +  INT (Z * BF ^ (L - X))
200  NEXT X
210  REM ----------------------
220  REM  CHANGE TO NEEDED BASE
230  REM ----------------------
240 TD = DEC
250 Q =  INT (TD / BT)
260 R =  INT ((TD / BT - Q) * BT + .5)
270 C$ = D$(R) + C$
280 TD = Q
290  IF TD > 0 THEN 250
300  PRINT "BASE "BF,"BASE "BT
310  PRINT "========","========"
320  PRINT N$,C$
330  END
340  DATA 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F
```

Appendix C

Sample Shape Table Codes

The vector paths and shape definitions for the shapes used in the Sample Shape Table are presented in Chapter 4. The vector and hexadecimal code conversions for the vector paths of the half note, treble clef, Nebraska, and maple leaf shapes are shown in Fig. C-1, C-2, C-3 and C-4.

HALF NOTE

VECTOR CODE CHART

SECTION								
3		2			1			
COLUMN								HEX
7	6	5	4	3	2	1	0	CODES
0	0	0	1	1	1	0	0	1C
0	0	1	1	1	1	1	1	3F
1	0	1	1	1	0	1	0	BA
0	0	1	1	0	1	1	0	36
0	0	0	0	1	1	1	0	0E
0	0	0	0	1	1	1	0	0E
0	1	1	0	0	1	0	1	65
0	0	0	0	1	1	0	0	0C
0	0	1	0	0	1	0	0	24
0	0	1	0	0	1	0	0	24
0	0	1	0	0	1	0	0	24
0	0	1	0	0	1	0	0	24
0	0	1	0	0	1	0	0	24
0	0	1	0	0	1	0	0	24
0	0	1	0	0	1	0	0	24
0	0	1	0	0	1	0	0	24
0	0	1	0	0	1	0	0	24
0	0	1	0	0	1	0	0	24
0	0	1	0	0	1	0	0	24
0	0	0	0	0	1	0	0	04

00 - END OF CODE

Fig. C-1. The vector code chart and the hex codes for the half note shape of the Sample Shapes shape table are presented in this figure.

VECTOR CODE CHART

SECTION								
3		2			1			HEX
COLUMN								
7	6	5	4	3	2	1	0	CODES
0	0	1	1	1	1	1	1	3F
0	0	0	1	0	1	1	1	17
0	0	1	0	1	1	0	1	2D
0	0	1	1	0	1	0	1	35
1	1	1	1	1	1	1	1	FF
0	0	1	1	0	1	1	0	36
0	0	0	0	1	1	1	0	0E
0	0	0	1	0	1	0	1	15
0	0	1	0	1	1	0	1	2D
0	0	0	0	1	1	0	0	0C
0	0	0	0	0	1	0	1	05
0	0	1	0	0	0	0	0	20
0	0	1	0	0	1	0	0	24
0	0	1	0	0	1	0	0	24
0	0	1	0	1	1	0	0	2C
0	0	1	0	1	0	0	0	28
0	0	1	0	1	0	0	0	28
0	0	1	0	1	1	0	0	2C
0	0	1	0	1	1	0	0	2C
0	0	1	0	0	1	0	0	24
0	0	1	0	0	1	0	0	24
0	0	1	0	0	1	0	0	24
0	0	1	0	0	1	0	0	24
0	0	1	0	0	1	1	1	27
0	0	1	0	0	1	1	1	27
0	0	1	1	1	1	1	1	3F
0	0	1	1	0	1	1	1	37
0	0	1	1	0	1	1	0	36
0	0	1	1	0	1	1	0	36
0	0	1	1	0	1	1	0	36
0	0	1	1	0	1	1	0	36
0	0	1	1	0	1	1	0	36
0	0	1	1	1	1	1	1	3F
0	0	0	0	0	1	1	1	07
0	0	1	1	1	0	0	0	38
0	0	0	0	0	1	1	1	07
0	0	1	0	0	0	0	0	20
0	0	1	1	1	1	1	1	3F
0	0	1	0	1	0	0	0	28
0	0	1	1	1	1	0	0	3C

VECTOR CODE CHART

SECTION								
3		2			1			HEX
COLUMN								
7	6	5	4	3	2	1	0	CODES
0	0	1	0	0	1	0	0	24
0	0	1	0	0	1	0	0	24
0	0	1	0	0	1	0	0	24
0	0	0	0	0	1	0	1	05
0	0	1	1	1	0	0	0	38
0	0	1	0	1	1	0	0	2C
0	1	1	0	0	1	0	0	64
0	1	1	0	0	1	0	0	64
0	0	0	0	1	1	0	0	0C
0	0	0	0	1	1	0	0	0C
0	0	0	0	1	1	0	0	0C
0	0	0	0	1	1	0	0	0C
0	0	1	0	0	1	0	0	24
0	0	1	0	0	1	0	0	24
0	0	1	0	0	1	0	0	24
0	0	1	0	0	1	0	0	24
0	0	1	0	0	1	0	0	24
0	0	1	0	1	1	0	0	2C
0	0	0	1	0	1	0	1	15
0	0	0	1	0	1	0	1	15
0	0	1	0	1	1	1	0	2E
0	0	1	1	0	1	1	0	36
0	0	1	1	0	1	1	0	36
0	0	1	1	1	1	1	0	3E
1	0	1	1	1	1	1	0	BE
0	0	0	1	1	1	1	0	1E
0	0	0	1	1	1	1	0	1E
0	0	1	1	0	1	1	0	36
0	0	1	1	0	1	1	0	36
0	0	1	1	0	1	1	0	36
0	0	0	1	0	1	1	1	17
0	0	0	1	0	1	1	1	17
0	0	0	1	0	1	1	1	17
0	0	1	1	0	1	1	0	36
0	0	1	1	0	1	1	0	36
0	0	0	0	1	1	1	0	0E
0	0	0	0	0	1	1	0	06

00 - END OF CODE

Fig. C-2. This figure shows the hex codes and vector code chart of the treble clef shape that appears in the Sample Shapes shape table.

NEBRASKA

VECTOR CODE CHART

SECTION								
3			2			1		
COLUMN								HEX
7	6	5	4	3	2	1	0	CODES
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	0	0	1	0	0	24
0	0	1	0	0	1	0	0	24
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	0	0	1	0	0	24
0	0	1	0	0	1	0	0	24
0	0	1	0	0	1	0	0	24
0	0	1	0	0	1	0	0	24
0	0	1	0	1	1	0	0	2C
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	1	0	1	0	1	35
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	1	0	1	1	0	36
0	0	1	1	0	1	0	1	35
0	0	1	1	0	1	0	1	35
0	0	1	0	1	1	1	0	2E
0	0	1	1	0	1	1	0	36
0	0	1	1	0	1	1	0	36
0	0	1	1	0	1	0	1	35
0	0	1	1	0	1	1	0	36

00 - END OF CODE

Fig. C-3. The vector code chart and hex codes for the Nebraska shape are shown here.

VECTOR CODE CHART

SECTION								
3			2			1		
COLUMN								HEX
7	6	5	4	3	2	1	0	CODES
0	0	1	0	0	1	0	1	25
0	0	0	0	0	1	0	1	05
0	0	1	0	0	0	0	0	20
0	1	1	0	0	1	0	0	64
0	0	1	0	0	1	0	0	24
0	0	1	0	1	1	0	0	2C
0	0	1	0	1	1	0	1	2D
0	0	1	1	0	1	0	1	35
0	0	1	0	0	1	0	1	25
1	0	1	1	0	1	0	1	B5
0	0	0	0	1	1	0	1	0D
0	0	1	1	1	0	0	0	38
1	0	1	0	1	0	0	0	A8
0	0	1	0	0	1	0	1	25
0	0	0	1	1	0	0	0	18
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
1	1	1	1	1	1	1	1	FF
1	1	0	1	1	1	1	1	DF
0	1	1	0	1	0	0	0	68
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	0	0	0	1	0	0	04
1	0	1	1	1	0	0	0	B8
1	1	1	1	1	1	1	1	FF
0	0	1	1	1	0	1	1	3B
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	0	0	0	1	1	1	07
0	1	1	0	0	0	0	0	60
0	0	1	0	1	1	1	0	2E
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D

VECTOR CODE CHART

SECTION								
3			2			1		
COLUMN								HEX
7	6	5	4	3	2	1	0	CODES
0	1	1	0	1	1	0	1	6D
0	0	1	1	1	0	0	0	38
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	0	1	1	0	0	2C
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	0	0	1	1	0	1	0D
0	0	1	1	1	0	0	0	38
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	0	1	0	0	0	28
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	1	1	0	0	0	38
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
1	1	0	1	1	1	1	1	DF
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	0	0	1	1	0	0	0C
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D

Fig. C-4. The maple leaf shape definition is represented in this figure.

VECTOR CODE CHART

SECTION								
3			2			1		
COLUMN								HEX
7	6	5	4	3	2	1	0	CODES
0	0	1	0	1	1	0	1	2D
0	1	0	0	1	1	0	1	4D
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	0	0	1	0	0	24
0	0	0	1	0	1	1	1	17
0	0	0	1	1	1	1	1	1F
1	1	0	1	1	1	1	1	DF
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	0	0	1	0	0	24
0	0	1	0	0	1	0	0	24
0	0	0	0	0	1	1	1	07
0	0	1	0	0	0	0	0	20
0	0	0	1	0	1	0	1	15
0	0	0	1	0	1	0	1	15
0	1	1	0	1	1	1	0	6E
0	0	1	1	1	1	1	0	3E
0	0	1	1	0	1	1	1	37
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	0	0	0	1	0	1	05
0	0	1	1	1	0	0	0	38
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	0	0	1	1	1	27
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	0	0	1	0	1	25
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F

VECTOR CODE CHART

SECTION								
3			2			1		
COLUMN								HEX
7	6	5	4	3	2	1	0	CODES
0	0	1	1	1	1	1	1	3F
0	0	1	0	0	1	1	1	27
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	0	1	1	1	0	0	1C
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	0	0	1	1	1	27
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	1	0	0	0	0	30
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	1	1	1	1	1	1	3F
0	0	0	1	1	1	1	1	1F
0	1	0	0	1	1	0	0	4C
0	0	1	0	0	1	0	0	24
1	0	0	1	0	1	0	1	95
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
1	1	1	1	1	1	0	0	FC
0	0	1	1	1	1	1	1	3F
0	0	1	0	0	1	1	1	27
0	0	1	0	1	1	0	1	2D
0	0	1	0	1	1	0	1	2D
0	0	1	1	1	0	1	1	3B
0	0	1	1	1	1	1	1	3F
0	1	1	0	0	1	1	1	67
0	0	1	0	1	1	0	1	2D
0	0	1	1	1	1	0	0	3C
0	0	0	0	1	1	0	0	0C
0	0	0	0	0	1	0	1	05

00 - END OF CODE

Fig. C-4 (concluded).

Appendix D

Computer Memory Structures

Information that is meaningful to the computer is represented using the binary digits 0 and 1. The memory structures which store the binary digits are called *bits*. The entire vocabulary of the computer can be expressed in 8-bit words. The 8-bit memory units used for storing the words are called *bytes*. The computer used to develop the programs presented in this book has 65,536 bytes of memory. A computer of this size is said to have *64 kilobytes* or *64K* of memory. Each kilobyte is actually 1024 bytes, despite the prefix *kilo,* which literally means 1000. All discussions in this book concerning memory are based on a 64K memory computer.

Although a byte of memory stores 8 bits of information, a series of binary digits like 10011100 is not easy for people to comprehend. Since any 8-digit base-2 number can be expressed as a 2-digit base-16 number, the information stored in a byte can easily be converted to hexadecimal notation. The binary number 10011100 is equivalent to the hexadecimal number 9C. The greatest numerical value that can be stored in one byte of memory is 11111111 (base-2), FF (base-16), or 255 (base-10). It is a general practice to represent the information stored in a memory byte in its hexadecimal form.

Whenever the computer is on, every byte of memory contains some kind of information. The computer must be able to access selectively each byte of memory in order to use the information stored in it. To satisfy this requirement, each byte is assigned a number or *address*. Addresses start with 0 and go up to 65535. Each address is represented by a 4-digit hexadecimal number from 0000 to FFFF. Figure D-1 illustrates a memory unit consisting of the address and content of a byte.

One useful application of the Number Base Conversion program listed in Appendix B is changing memory addresses from base-10 to base-16, or vice versa.

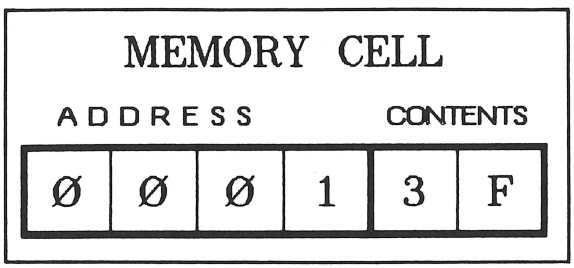


Fig. D-1. Each byte of memory holds a two-digit hex number, but its address is represented by a four-digit hex number.

Appendix E

Computer Memory Utilization

Computers have two different types of memory. *Read-only memory (ROM)* is comprised of bytes of memory which cannot be altered by the programmer; it contains the instructions that tell the computer how to operate, as well as the BASIC programming language. The *random access memory (RAM)* bytes can be changed by the programmer. RAM is the memory where BASIC programs and the variable values are stored.

RAM memory is divided into several segments or *address ranges*. The roles reserved for information stored in these segments are most easily expressed with a memory map. The memory map in Table E-1 shows the major address ranges of the memory segments and the roles associated with information stored in the segments.

The standard RAM memory utilization is occasionally not suitable for programs that use high-resolution graphics. In situations which require special memory considerations, the commands HIMEM, LOMEM, and MAXFILES can be used to reallocate the role of RAM memory segments.

The HIMEM statement establishes the highest memory address that a BASIC program can use for storing strings. HIMEM is set by the computer whenever the system is booted to a default value of 36863, which is just below the Disk Operating System (DOS) instruction set. Alternate HIMEM values are necessary when the programmer needs to store information in the area of memory normally allocated for string storage. By using a statement like HIMEM:32768, 4095 bytes of memory are shifted from string storage to programmer usage.

LOMEM, meaning low memory, can be used to specify the lowest memory address which BASIC is to use for storing variables. Variable values are normally stored beginning with the next byte beyond the end of the BASIC program. The statement LOMEM:24576 is sometimes used to prevent the high-resolution graphics segments of memory from being used for variable storage.

Some programs are either very long or use lots of variables or strings. If the amount of memory needed by a program is greater than that available, some of the memory allocated to DOS file buffers can be freed for other purposes through the use of

Table E-1. Typical Memory Map for 48K RAM Computer.

ADDRESS RANGE		MEMORY UTILIZATION
Decimal	Hexadecimal	
0 - 511	0000 - 01FF	Program workspace (computer use).
512 - 767	0200 - 02FF	Input buffer.
768 - 975	0300 - 03CF	Free space for programmers' use.
976 - 1023	03D0 - 03FF	DOS jump pointers.
1024 - 2047	0400 - 07FF	Text and Low Res Graphics Screen.
2048 - 38399*	0800 - 95FF*	Programs are stored from here to the end of the program, HIMEM, or DOS; whichever is first.
8192 - 16383	2000 - 3FFF	High Resolution Graphics Page 1.
16384 - 24575	4000 - 5FFF	High Resolution Graphics Page 2.
38400 - 49151	9600 - BFFF	Disk Operating System [DOS].
49152 - 53247	C000 - CFFF	Hardware I/O addresses.
53248 - 65635	D000 - FFFF	Read Only Memory [ROM].

* Highest address available for program storage based on normal configuration. This area is shared by the program, variables, and strings as well as the high resolution graphics pages.

the MAXFILES command. This command is not used in any programs in this book; it is more commonly used in programs that involve the use of text files. The maximum number of DOS buffers normally set when DOS is booted is 3. Input and output (I/O) buffers of 595 bytes are reserved for use with each of the files. If no files are used within the program, the statement MAXFILES 1 will free up approximately 1180 additional bytes of memory for other purposes. HIMEM is set to the appropriate address automatically when MAXFILES is used.

Nearly every programmer has either seen or written programs which use the commands PEEK, POKE, and CALL. They are very powerful commands that allow the programmer to directly access specific bytes of the computer's memory.

The contents of any memory byte can be examined by using the PEEK command. A statement like

```
X=PEEK(128)
```

causes the number stored in memory location 128 to be assigned to the variable X. Both the address parameter and the resultant value are expressed as decimal numbers. The address parameter of the PEEK command may not exceed 65535. The statement

```
PRINT PEEK (-5)
```

causes the contents of byte 65531 to be output on the screen of the monitor. Any byte address can be expressed as a negative number by subtracting

Table E-2. Common BASIC-Accessed Memory Addresses.

ADDRESS POS	ADDRESS NEG	DESCRIPTION	FUNCTION PEEK	FUNCTION POKE	FUNCTION CALL
32	-65504	CHECK/SET LEFT EDGE OF TEXT WINDOW	*	*	
33	-65503	CHECK/SET TEXT WINDOW WIDTH	*	*	
34	-65502	CHECK/SET TOP LINE OF TEXT WINDOW	*	*	
35	-65501	CHECK/SET BOTTOM LINE OF TEXT WINDOW	*	*	
36	-65500	CHECK/SET HORZ POSITION OF CURSOR	*	*	
37	-65499	CHECK/SET VERTICAL POSITION OF CURSOR	*	*	
216	-65320	ERROR CODE	*		
222	-65314	RESET ONERR FOR NORMAL MESSAGES		*	
230	-65306	HIGH RESOLUTION PLOTTING PAGE		*	
232	-65304	LOW BYTE OF SHAPE TABLE ADDRESS		*	
233	-65303	HIGH BYTE OF SHAPE TABLE ADDRESS		*	
234	-65302	COLLISION COUNTER	*		
49152	-16384	READ KEYBOARD	*		
49168	-16368	CLEAR KEYBOARD STROBE		*	
49200	-16336	CLICK THE SPEAKER	*		
49232	-16304	SET DISPLAY TO GRAPHICS MODE		*	
49233	-16303	SET DISPLAY TO TEXT MODE		*	
49234	-16302	SET FULL SCREEN GRAPHICS		*	
49235	-16301	SET MIXED GRAPHICS/TEXT SCREEN		*	
49236	-16300	DISPLAY PAGE 1 ON SCREEN		*	
49237	-16299	DISPLAY PAGE 2 ON SCREEN		*	
49238	-16298	DISPLAY TEXT PAGE		*	
49239	-16297	DISPLAY HIGH RESOLUTION SCREEN		*	
49249	-16287	PADDLE 1 BUTTON PRESSED IF >127	*		
49250	-16286	PADDLE Ø BUTTON PRESSED IF >127	*		
62450	-3086	CLEAR CURRENT HIGH RESOLUTION SCREEN			*
62454	-3082	SET HIGH RESOLUTION SCREEN TO COLOR OF LAST PLOTTED POINT			*
64578	-958	CLEAR SCREEN FROM CURSOR TO BOTTOM OF SCREEN			*
64600	-936	SAME AS HOME IN BASIC			*
64668	-868	CLEAR SCREEN FROM CURSOR TO END OF LINE			*
65338	-198	BELL			*
65385	-151	ENTER APPLE MONITOR			*

65536 from the positive address of the byte. The specific roles of several byte addresses that pertain to graphics programming are discussed throughout the book.

The POKE statement is used to write specific information into a particular byte of memory. If the number 6 is to be placed in the memory byte with the decimal address 256, the statement

```
POKE 256,6
```

would be used. The greatest address parameter of the POKE statement is 65535. The greatest number that can be stored in a single byte of memory is 255. Only decimal numbers can be used with the POKE statement. Negative addresses may be used in the POKE statement. Refer to the previous paragraph for more information on negative byte addresses.

CALL statements allow programmers to use machine language subroutines from within a BASIC program. Either subroutines written by the programmer or those residing in ROM may be accessed. The statement

```
CALL -198
```

has the same effect (bell sound) as pressing CTRL-G on the keyboard and can be used in a program instead of the statement PRINT CHR$(7).

Table E-2 is a reference chart containing commonly used PEEK, POKE, and CALL addresses and their roles.

Appendix F
Summary of Programs

The programs presented in each chapter are listed below, along with a short description of each.

CHAPTER 2 PROGRAMS

Low Res Commands—Demonstrates most of the low-resolution graphics commands.

Purple Boat—Displays a small purple sailboat design in low-resolution graphics.

Reference Point Boat—Illustrates how to utilize the concept of describing designs with respect to a reference point.

Three Boats—Displays three boats on the screen using the reference point technique.

Rainbow Boat—Displays the sailboat using several colors.

Random Plotter—Demonstrates how to use the random function to plot low-resolution color graphics.

Color Lines—Randomly places low-resolution graphics lines on the screen.

Patches—The Color Lines program was modified to create a patchwork design.

Sketcher—Allows users to create low-resolution graphics displays from the keyboard.

Paddle Sketcher—Modifications were made to the Sketcher program so that displays could be created using paddles rather than the keyboard.

Buzzer—A short program demonstrating how to use the computer speaker to generate a buzzing sound.

Clickity Plot—Randomly places points on the screen, producing a click sound as each point is plotted.

Pink Out—A computerized board game which utilizes most of the programming techniques discussed in Chapter 2.

CHAPTER 3 PROGRAMS

Color Sweep—Draws high-resolution graphics lines from the center of the screen to points along the edge of the screen.

Rectangle—Demonstrates the use of the HPLOT-TO command to construct a rectangle.

General Rectangle—Utilizes the reference point technique to display a rectangle on the high-resolution graphics screen.

T-FORM—Demonstrates the use of subroutines to create a high-resolution graphics display.

Scalar T—Introduces a technique for scaling high-resolution graphics designs.

Robot—Utilizes a rectangle scaling subroutine to display a robot.

Solid Rectangle—Illustrates how to display solid designs.

Hourglass Timer—Sand is sifted through a high-resolution graphics hourglass.

Equation Grapher—Program can be used to display the graphs of most mathematical equations.

Circle—Introduces the concept of using polar coordinates to create a high-resolution graphics circle design.

Ellipse—Modifications to the Circle program were made for displaying ellipses.

Circle/Ellipse Subroutine—This subroutine can be placed in any program for displaying circles and ellipses on the high-resolution graphics screen.

One-Eye—Demonstrates how to use the Circle/Ellipse subroutine.

Winter Word Game—A word guessing game that utilizes many of the programming concepts presented Chapter 3.

Five-Petal Rose—Program is used to graph polar equations.

Polygons—Displays polygons with up to 12 sides on the high-resolution graphics screen.

Solid Polygons—Demonstrates the advanced techniques needed to display solid polygons.

Polar Design Maker—Creates complex designs based on polar equations and saves them in binary data files on the disk.

Display Design—Demonstrates a simply way to display high-resolution graphics designs that have been saved on the disk.

No-Fad Display—Loads existing pictures into the high-resolution graphics page 2 memory from the disk and displays them on the screen.

Page 1 Display—Similar to No-Fade Display except graphics are displayed on high-resolution graphics page 1.

Alternating Designs—Demonstrates the page flipping technique.

Slide Show—Displays high-resolution graphics pictures that have been stored on the disk. The user controls the rate at which the pictures are displayed.

CHAPTER 4 PROGRAMS

Sample Shapes—This is a shape table developed in Chapter 4 and used in many of the programs presented in Chapter 5.

CHAPTER 5 PROGRAMS

Debut—Demonstrates the commands needed to display shapes on the high-resolution graphics screen.

Color Trial—Allows the user to observe the effects of using different colors to display shapes on different background colors.

Scale and Rotation—Demonstrates the relationships between the SCALE and ROT commands.

Motion—Introduces the techniques used to cause shapes to appear to move across the screen.

Character Set—This shape table is used for displaying text characters on the high-resolution graphics screen.

Draw 1 Character—Displays one character from the Character Set shape table.

Character Display—Displays all the shapes from the Character Set shape table.

Keyboard Graphics—Allows the user to place shapes on the screen by pressing the keys which correspond to the characters in the shape table.

Vertical Keyboard—Modifications were made to the Keyboard Graphics program which causes it to place rotated characters on the screen.

Mixer—A partial program that illustrates the techniques of mixing text shape characters and standard high-resolution graphics designs on the screen. The user's response to a question is placed on the screen as it is typed. The response is evaluated, and an appropriate message is displayed.

Alternate Mixer—A variation of the Mixer program, modified for faster execution.

Game Shapes—A shape table designed to be used with the Arcade Space Game.

Arcade Space Game—A simple game that demonstrates some techniques that can be used in creating arcade-style games.

Draw-N-Flip—Demonstrates how page flipping can be used to improve motion sequences involving shape tables.

CHAPTER 6 PROGRAMS

Table Developer—A utility program that can be used to create shape tables containing up to 255 shapes.

Table Merger—Allows users to copy shapes from one shape table into another. Both shape tables must be created with the Table Developer program.

Shapely Scenes—This utility program allows users to create either high-resolution pictures or action sequences from shapes stored in a shape table.

Scene Viewer—Displays scenes created with the Shapely Scenes program.

Letter Scene—This is a scene data file which demonstrates a type of action scene that can be created with the Shapely Scenes program.

CHAPTER 7 PROGRAMS

Note Player—A machine language program which creates note sounds.

Note Tester—This program demonstrates how the Note Player program can be used in a BASIC program.

Random Notes—Selects random pitches that are played by the Note Player program.

Comb Sound—Sequences of short notes are played to create a sound similar to the sound produced by raking a comb across the edge of a table.

Random Comb Song—A variation of the Comb Sound program that randomly selects the range of notes used in creating the sounds.

Two Combs—Another variation of the Comb Sound program which appears to generate two comb raking sounds at once.

Random Comb Duet—The concepts used in the Random Comb Song and the Two Combs programs were combined to generate the sounds produced by this program.

Musical Scale—This program utilizes the Note Player program to play the notes of the scale.

Fanfare Song—Demonstrates how to use the Note Player program to play a song.

Musical Sounds—A machine language program that can play an entire song, rather than just one note (as is the case with the Note Player program).

Song Editor—Creates song data files that can be played by the Musical Sounds program.

Laser Down—A machine language program producing a tone that rapidly decreases in pitch.

Laser Up—A machine language program producing a tone that rapidly increases in pitch.

Random Laser—Randomly produces sounds using the Laser Up program. This program demonstrates that the laser sound programs can be used to generate a variety of interesting sound effects.

CHAPTER 8 PROGRAMS

One Vector—A shape table containing a single plotting vector shape definition. Several programs in this chapter are used to demonstrate the diverse applications of such a shape.

Vector Circle—Uses the One Vector shape to display a circular design.

Vector Spiral—Creates a spiral design using the One Vector shape.

Vector Spin—Demonstrates a rotary motion technique.

Vector Spiral Spin—Applies the technique for spinning motion to the display created by the Vector Spiral program.

Vector Donut—Demonstrates using a single vector shape to create a donut design.

Vector Polygons—Illustrates how to use a single vector to display polygon figures.

Arrows—The shapes in this shape table are used

in several of the remaining programs in Chapter 8.

Arrow Spin I—Demonstrates spinning motion with two shapes from the Arrow shape table.

Arrow Spin II—Similar to the Arrow Spin I program. Uses a single scaled vector to extend the shaft of the arrow.

Arrow Orbit—Instead of using a scaled vector shape to extend the shaft of an arrow, this program uses the scaled vector to create an orbiting motion sequence.

Wreath—Demonstrates that shapes can be used to create artistic displays that do not resemble the shapes themselves.

Cylinder—A cylinder design is created using the One Vector shape table.

Cylinder Sweep—Adds a motion sequence to the Cylinder program to create a rocking illusion.

Fan-O-Rama—This program creates an interesting action display which incorporates sound effects.

Shapely Ribbon—Sound, shapes and polar equations are used in forming the display created by this program.

Directed Rose—In addition to the techniques used in the Shapely Ribbon program, Directed Rose utilizes page flipping to create an unusual display.

Fraction Action—This is a sophisticated instructional program which employs most of the techniques presented in Chapters 3 through 8 to drill students on the addition of fractions.

CHAPTER 9 PROGRAMS

Compare Motions—A machine language program that demonstrates motion sequences in machine language. The program allows users to compare motion accomplished with and without page flipping.

Motion Comparison—This program uses the Compare Motions program to illustrate the difference between motion sequences in BASIC and those in machine language.

Raster Motion—Raster graphics (bit mapping) are used in this machine language program to demonstrate the tremendous speed advantage to this method of displaying high resolution graphics.

APPENDIX B PROGRAM

Number Base Conversion—This utility program provides programmers an easy way to convert numbers from one base (radix) to another.

Index

Index

A

ABS function, 37
Absolute value, 37
Address, memory, 184
Address range, 185
Algorithm, 51
Alternate Mixer program, 95
Alternate Mixer program, enhancement module for, 96
Alternating Designs program, 56
Apple II, display modes of, 3
Arcade Space Game program, 98
Arcade Space Game shape table, 97
Arcade-style programs, 96
Argument, function, 9
Arrow Orbit program, 150
Arrow Spin I program, 148
Arrow Spin II program, 149
ASCII, 13, 174
Assembler program, 124

B

Binary file, 52
Bit (binary digit), 184
BLOAD command, 55
BSAVE command, 54
Buzzer program, 15
Byte, 184

C

CALL command, 186
Cartesian coordinates, 35
CHAIN (utility program), 57
Character Display program, 87
Character Set shape table, hexadecimal listing of, 84
Circle/ellipse subroutine, 41
  demo program for, 42
Circle plotting program, 40
Clickity Plot program, 15
COLOR instruction, graphics, 4
Color Lines program, 11
Color Sweep program, 28
Color Trial program, 78
Colors
  high-resolution, table of, 27
  low-resoluton, random selection of, 10
  low-resolution, table of, 4
Comb Sound program, 127
Compare Motions program
  assembly listing, 166
  hexadecimal listing, 169
Convention, notation, 2
Coordinate
  cartesian, 35
  horizontal, 4
  ordered pair, 4
  polar, 39
  rectangular, 35
  screen, 4
  vertical, 4
Cosine function, 37
CTRL characters, quotation marks with, 15
Cylinder program, 151
Cylinder Sweep program, 152

D

Debut program, 75
Deferred mode, 6
Degree/radian conversion chart, 40
Directed Rose program, 155
Disassembler program, Apple, 125
Display Design program, 55
Display mode
  high-resolution, 3
  low-resolution, 3
  screen, 3
  text, 3
DIV function, simulating in Applesoft, 137
DOS command
  deffered-mode, 76
  use in Applesoft, 54
Draw 1 Character program, 87
DRAW instruction, 75
Draw-N-Flip program, 102

E

Ellipse plotting program, 41
Ellipse subroutine, 41
Equation Grapher program, 37

Equation
graphic, table of forms, 38, 49
polar, 49
straight-line, 51
Equipment, required, 2
Execution
deferred, 6
immediate, 6

F

Fan-O-Rama program, 153
Fanfare Song program, 130
File, binary, 52
Five Petal Rose program, 48
Fraction Action program, 156
Function
argument of, 9
cyclic, 37
period of, 37

G

Game design, 16
General Rectangle program, 30
GET statement, 6
GR instruction, graphics, 4
Graph paper, use in planning, 7
Graphics
applications of, 14, 42
enhancements of, 149
high-resolution, 3, 26
input and output displays
low-resolution, 3
low-resoluton, instruction set for, 4
mixing with text, 82
page flipping, 26
planning, 7
plot, 3, 149
random displays, 9
raster, 171
screen management, 29
use of polygons, 30
use of reference point, 8
use of sound with, 15
vector, 59

H

HCOLOR instruction, graphics, 26
HGR instruction, graphics, 26
HGR2 instruction, graphics, 26
High-resolution graphics, properties of, 27
HIMEM: command, 104, 185
HLIN instruction, graphics, 4
HOME instruction, 6
Hourglass Timer program, 34
HPLOT instruction, graphics, 26
HPLOT-TO instruction, graphics, 27

I

Immediate mode, 6
Index, shape table, 69
Infinite loop, termination of, 11
Instruction set
high-resolution, 26
low-resolution, 4
INT function, 10
Irrational number, 37

K

Kilobyte, 184

L

Laser Down program
assembly listing, 140
hexadecimal listing, 141
Laser Up program
assembly listing, 141
hexadecimal listing, 141
Letter Scene shape table, 121
Line
slope of, 51
y-intercept of, 51
LOMEM: command, 185
Loop
infinite, 11
time delay, 6
Low-Res Commands program, 6

M

Machine language, 125-165
MAXFILES: command, 185
Memory move command, Monitor, 106
Memory
random access, 185
read-only, 185
Mixer program, 92
MOD function, simulating in Applesoft, 137
Monitor program, Apple, 72
Motion Comparison BASIC listing, 170
Motion
simulation with shape table, 80
spinning, 146
Music, creation of, 127
Musical Scale program, 129
Musical Sounds program, assembly listing, 133
Musical Sounds program, hexadecimal listing, 134
Musical symbols, table of, 132

N

Name, variable, 2
No-Fade Display program, 55
Notation convention, 2
Notation, musical, 132
NOT logical operator, 47
Note pitch table, 128
Note Player program
assembly listing, 125
hexadecimal listing, 125
Note Tester program, 126
Number Base Conversion program, 177
Number system
binary, 176
decimal, 176
hexadecimal, 176
Number
hexadecimal, 54
irrational, 37
random, Apple II, 9

O

Offset, address, 73
One-Eye program, 42
Opcode, assembly language, 124
Origin, rectangular coordinates, 35

P

Paddle Sketcher program, 14
Page, memory, 27
Page 1 Display program, 56
Page flipping, 26, 52, 56
Patches program, 11
PDL function, 14
PEEK command, 186
Period, of function, 37
Pink Out program, 19
Pixel, 3
Plot graphics, 3
PLOT instruction, graphics, 4
Point, low-resolution, size of, 4
POKE command, 186
Polar Design Maker program, 53
Polar equation, 49
Polygon
regular, 49
vertices of, 30
with graphics, 30
Polygons program, 50
POP statement, 48
Program, arcade-style, 96
Purpose Boat program, 8

Q

Quadrant, rectangular coordinates, 35
Quotation marks, with CTRL characters, 15

R

Radian, 39
Radian/degree conversion chart, 40
Rainbow Boat program, 9
Random access memory (RAM), 185
Random Comb Duet program, 128
Random Comb Song program, 127
Random Laser program, 142
Random Notes program, 126
Random numbers, Apple II, 9
Random Plotter program, 10
Raster graphics, 171
Raster Motion, hexadecimal listing, 173

Raster Motion program, assembly listing, 172
Read-only memory (ROM), 185
Rectangle program, 30
Rectangular coordinates, 35
Reference point, graphics, 8
Reference Point Boat program, 8
Required equipment, 2
Resoluton, 3
Resolution, screen, 3
RND function, 9
Robot program, 32
ROT instruction, 75

S

Scale and Rotation program, 79
SCALE instruction, 175
Scaling variable, 32
Scene Viewer program, 120
Screen coordinate, 4
Screen management, graphics, 29
Screen
  low-resolution, 3
  low-resolution, dimensions of, 4
Shape definition, 59, 68
Shape definiton, stacking, 69
Shape table, 59
  capacity of, 69
  commands for, 75
  constructing, 69
  demo programs for, 77
  entering, 72
  index of, 69
  indexing for expansion, 107
  memory structure and selection, 104
  saving, 73
  simulating motion, 80
  text character, 82
Shape
  character set, 86
  defining, 60
  machine language display of, 165
  rotation of, 77
  screen display of, 74
  shading, 60
Shapely ribbon program, 154
Shapely Scenes program, 114
Sine function, 37
Sketcher program, 12
Slid Show program, 57
Slope, 51
Solid Polygons program, 51
Solid Rectangle program, 33
Song Editor program, 135
Sound, 124
  arcade game, 140
  with low-resolution grahics, 15
Speaker switch, memory location of, 15

T

Table Developer utility program, 109
Table Merger utility program, 113
TEXT instruction, 5, 27
TEXT instruction, graphics, 5
Text, mixing with graphics, 82
T-Form program, 31
Three Boats program, 9
Time delay loop, 6
Two Combs program, 128

V

Variable, scaling, 32
Variable name, 2
Vector, coding, 61
Vector Circle program, 144
Vector code, charting, 61
Vector Donut program, 146
Vector graphics, enhancement of, 149
Vector Polygons program, 146
Vector Spin program, 145
Vector Spiral program, 144
Vector Spiral Spin program, 145
Vertical Keyboard program, 91
VLIN instruction, graphics, 5

W

Wreath program, 150

X

XDRAW instruction, 75, 145

OTHER POPULAR TAB BOOKS OF INTEREST

The Computer Era—1985 Calendar Robotics and Artificial Intelligence (No. 8031—$6.95)
Programming with dBASE II® (No. 1776—$16.50 paper; $26.95 hard)
Lotus 1-2-3™ Simplified (No. 1748—$10.25 paper; $16.95 hard)
Mastering Multiplan™ (No. 1743—$11.50 paper; $16.95 hard)
Apple Logo for Kids (No. 1728—$11.50 paper; $16.95 hard)
Getting the Most from Your Pocket Computer (No. 1723—$9.95 paper; $14.95 hard)
Scuttle the Computer Pirates: Software Protection Schemes (No. 1718—$15.50 paper; $21.95 hard)
Computer Programs for the Kitchen (No. 1707—$13.50 paper; $18.95 hard)
Beginner's Guide to Microprocessors—2nd Edition (No. 1695—$9.95 paper; $14.95 hard)
Apple® Lisa™: A User-Friendly Handbook (No. 1691—$16.95 paper; $24.95 hard)
How To Create Your Own Computer Bulletin Board (No. 1633—$12.95 paper; $19.95 hard)
Does Your Small Business Need a Computer? (No. 1624—$10.95 paper; $18.95 hard)
Forecasting On Your Microcomputer (No. 1607—$15.50 paper; $21.95 hard)
BASIC Computer Simulation (No. 1585—$15.50 paper; $21.95 hard)
Solving Math Problems in BASIC (No. 1564—$15.50 paper; $19.95 hard)
Learning Simulation Techniques on a Microcomputer Playing Blackjack and Other Monte Carlo Games (No. 1535—$10.95 paper; $16.95 hard)
Basic BASIC-English Dictionary for the Apple™, PET®, and TRS-80™ (No. 1521—$17.95 hard)
Making Money with Your Microcomputer (No. 1506—$8.25 paper; $13.95 hard)
Investment Analysis with Your Microcomputer (No. 1479—$13.50 paper; $19.95 hard)
The Art of Computer Programming (No. 1455—$10.95 paper; $16.95 hard)
Using and Programming the Macintosh™, including 32 Ready-To-Run Programs (No. 1840—$12.50 paper; $16.95 hard)
Making CP/M-80® Work for You (No. 1764—$9.25 paper; $16.95 hard)
Going On-Line with Your Micro (No. 1746—$12.50 paper; $17.95 hard)
The Master Handbook of High-Level Microcomputer Languages (No. 1733—$15.50 paper; $21.95 hard)
How to Document Your Software (No. 1724—$13.50 paper; $19.95 hard)
101 Programming Surprises & Tricks for Your Apple II®/III®e Computer (No. 1721—$11.50 paper)
Interfacing and Digital Experiments with Your Apple® (No. 1717—$15.50 paper; $21.95 hard)
MicroProgrammer's Market 1984 (No. 1700—$13.50 paper; $18.95 hard)
PayCalc: How to Create Customized Payroll Spreadsheets (No. 1694—$15.50 paper; $19.95 hard)
The First Primer of Microcomputer Telecommunications (No. 1688—$10.25 paper; $14.95 hard)
33 Adult Computer Games in BASIC for the IBM PC®, Apple II®/III®e and TRS-80™ (No. 1627—$13.50 paper; $18.95 hard)
Microcomputers for Lawyers (No. 1614—$14.50 paper; $19.95 hard)
Computer Companion for the Apple II®/Apple //®e (No. 1603—$10.25 paper)
Database Manager in MICROSOFT® BASIC (No. 1567—$12.50 paper; $18.95 hard)
Troubleshooting and Repairing Personal Computers (No. 1539—$14.50 paper; $19.95 hard)
25 Graphics Programs in MICROSOFT® BASIC (No. 1533—$11.50 paper; $17.95 hard)
Apple II® BASIC (No. 1513—$13.50 paper; $19.95 hard)
The Handbook of Microprocessor Interfacing (No. 1501—$15.50 paper; $21.95 hard)
From BASIC to Pascal (No. 1466—$11.50 paper; $18.95 hard)
Computer Peripherals That You Can Build (No. 1449—$13.95 paper; $19.95 hard)

TAB TAB BOOKS Inc.

Blue Ridge Summit, Pa. 17214

Send for FREE TAB Catalog describing over 750 current titles in print.

ISBN 0-8306-0849-4

HI-RES GRAPHICS FOR THE APPLE® II/IIe/IIc TICE & STEVENS TAB 1849